Cryptographic Protocols in a Post-Quantum World

Fiona Johanna Weber

Email: crypto@fionajw.de
Website: https://fiona.onl

A catalogue record is available from the Eindhoven University of Technology Library

ISBN: 978-90-386-6282-4

Printed by ADC Dereumaux, s'Hertogenbosch, Netherlands.
Cover design by Fiona Johanna Weber

Cryptographic Protocols in a Post-Quantum World

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de Technische Universiteit Eindhoven, op gezag van de rector magnificus prof.dr. S.K. Lenaerts, voor een commissie aangewezen door het College voor Promoties, in het openbaar te verdedigen op maandag 3 februari 2025 om 13:30 uur

door

Fiona Johanna Weber

geboren te Garmisch-Partenkirchen, Duitsland

Dit proefschrift is goedgekeurd door de promotoren en de samenstelling van de promotiecommissie is als volgt:

Voorzitter:	prof.dr.ir. B. Koren
1e Promotor:	dr. A.T. Hülsing
2e Promotor:	prof.dr. T. Lange
Leden:	prof.dr. K. Paterson (ETH Zürich)
	prof.dr. T. Jager (Bergische Universität Wuppertal)
	dr.ir. L.A.M. Schoenmakers
	prof.dr. P. Schwabe (Radboud Universiteit Nijmegen)
	dr. B. Dowling (King's College London)

Het onderzoek of ontwerp dat in dit proefschrift wordt beschreven is uitgevoerd in overeenstemming met de TU/e Gedragscode Wetenschapsbeoefening.

Thanks

First I would like to thank my supervisor Andreas Hüllsing for his help and advice during the entire time and without which this thesis would not have been possible. But I am especially grateful for his great support and patience during the last two years, when I was dealing with my transition. The room that I received to deal with many of the associated issues was more than what I would have received elsewhere and it meant *a lot* to me!

I thank Tanja Lange for everything she did to give me the opportunity to write this thesis, her professional and personal support during the research phase and her extensive comments at the end.

I thank Benjamin Dowling, Tibor Jager, Kenneth Paterson, Berry Schoenmakers, and Peter Schwabe for all their work on my committee.

I thank my co-authors Yawning Angel, Kai-Chun Ning, Philip Zimmermann, and of course the aforementioned Andreas, Benjamin, Peter, and Tanja for their participation in our joined work. In the same vein I wish to thank Trevor Perrin and Denisa Greconici for their help with Post-Quantum Noise.

I thank all of my current and former colleagues for all the nice time we had together, both at work and in our free time.

I thank my mother and my father for everything they have ever done for me, there is simply too much to list. I thank my stepfather for continuing my father's legacy and accepting and supporting me ever since he entered my life.

I thank all of my friends for being there for me when you were and for all the times we spent together. There are so many people here that I wont state names, because any such list would inevitably be incomplete.

I thank everyone who helped and supported me with my transition. There are too many people to list, though I wish to name some of them explicitly, because they provided me with the most crucial of support at the hardest of times. Christel, for being there for me from the very beginning, Amber, for her advice on how to deal with the Dutch healthcare system, Steph, who was an invaluable aid early on, Sarah, for important practical advice, Maru, who provided me with material support when I needed it most, Mitchell, for crucial technical support, and very much Sofie, for helping me to understand myself.

And last, but very much not least Senna, my dear girlfriend who massively lifted my spirits over the last few months and helped me to finally discover what life is supposed to be like.

Contents

1. Introduction

The desire to transmit information in ways that prevent parties other than the intended receivers to learn meaningful information dates back all the way to antiquity. By the time of the second world war encryption was in common use by all major belligerents and important enough that the then not publicly known compromise of the German Enigma is widely accepted to have significantly benefited the allied war efforts. Harry Hinsley, a historian at Bletchley Park, famously conjectured that the break shortened the war "by not less than two years and probably by four years" [Hin93].

Use of cryptography outside of governmental organizations was however still fairly limited and certainly did not reach the point where the private communication of the average person was protected by it.

The development of personal computing devices and the public discovery of asymmetric cryptography created the conditions necessary to change this: In 1994 Netscape published the first version of their "Secure Sockets Layer", better known as "SSL" which became the basis for secure communication in the World Wide Web. Initially used primarily for online payments, the reveal of US mass surveillance programs by Edward Snowden massively increased the already existing trend towards using SSL (now renamed to "Transport Layer Security" or TLS) on almost the entire Web. But the Web was not the only place that saw increasing amounts of encryption: Chat software such as Off-the-Record-Messaging and Signal that gave even stronger guarantees for the security of their communication saw increasing usage, which eventually led to some of their major competitors adopting their security-protocols.

That said, this paradise of secure communication has been on numbered days from the early start: In 1994 Shor discovered that quantum computers can be used to efficiently factor large numbers and compute so called discrete logarithms [Sho94]. Almost all currently deployed asymmetric cryptography requires the hardness of one of these two problems; the only thing that prevents a catastrophic break is therefore the assumed current lack of sufficiently powerful quantum computers to run these attacks. When such a quantum computer will be available or if it is even possible to build one is an open question. An open question whose answer may importantly not become public knowledge until significantly after such a computer is built.

The good news here is that it appears likely that no such computer exists at the time of this writing. The bad news is that this only improves the situation marginally: While it is indeed impossible to impersonate a party today with a future quantum computer, it is very straightforward to decrypt today's communication with such a future quantum computer. This means that the permissible timeframe for transitioning to cryptographic algorithms that resist quantum computers, so called *Post Quantum Cryptography*, is not so much set by the arrival of efficient quantum computers, but by that point in time, minus the time that the encrypted information has to remain confidential and minus the time the actual transition takes.

The concerning news is, that it is not clear that we have not already passed this point.

1.1. NIST's Post-Quantum Competition

As of 2024 the National Institute of Standards and Technology (NIST) of the United States is running a competition[1] to select such algorithms for standardization. While NIST standards are strictly speaking not applicable outside the USA, they are often very influential and provided a competition-framework that was widely accepted as *the* place to select algorithms from. This went so far that the German BSI announced even before the end of the competition that it considered two of the candidates (namely "Classic McEliece" [ABC+22] and "Frodo" [NAB+20]) good enough for immediate use before any winners were even announced by NIST. The French ANSSI and the Dutch AIVD did the same shortly after.

After the third round of the post-quantum competition, NIST announced on the fifth of July 2022 a single Key-Encapsulation-Mechanism (KEM) and three signature schemes as winners: In the former category this was Crystals-Kyber [SAB+22], an overall fairly compact and very high performing scheme. In the latter Kyber's sister scheme Crystals-Dilithum [LDK+22] was the primary winner due to its overall decent characteristics. Additionally, NIST also selected the notoriously hard to implement but more compact Falcon [PFH+22] for use cases in which small signatures are important and the much worse performing and (with regards to signatures) significantly larger, but extremely conservative SPHINCS+[HBD+22] for when a very conservative scheme is needed.

[1] https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization/Call-for-Proposals

Beyond these four schemes NIST continued the competition with a fourth round for KEMs to choose a fallback for Kyber that is based on different assumptions. The four candidates for this round are:

- The aforementioned Classic McEliece [ABC+22] which has humongous public keys and a fairly slow key-generation-process, but otherwise outstanding performance and size characteristics,
- Bike [ABB+22] and HQC [AAB+22], two schemes that are like McEliece based on error-correcting codes and have similar performance as Frodo (everything about 10 times larger than Kyber) which most notably means a much smaller public key than Classic McEliece has,
- and, technically still SIKE [JAC+22] which is based on so called supersingular elliptic curves but has been comprehensively broken citeEPRINT:CasDec22 since it was selected for the fourth round.

Besides these NIST schemes there are already international standards for so called stateful signatures that are (like SPHINCS+) only based on the security of hash functions: XMSS(-MT) [HBG+18] and LMS [MCF19]. Here "stateful" means that every signing operation causes an update to the secret key; reusing an already used state can result in catastrophic breaks of the scheme and is widely considered to be a sizeable "foot-gun" (e.g. [Lan13]). The reasonable performance and the extremely conservative security of these schemes can however make their use a worthwhile trade-off in situations with manageable risk of state reuse.

Lastly NIST announced a further competition for more post-quantum signature schemes that is in its second round as of 2024 and whose results will not be available for several years.

For a more comprehensive discussion of the schemes involved in the NIST competition we refer to Section 6.5.

1.2. Protocols

While secure primitives are usually a necessary condition to achieve secure communication, they are not a sufficient one: We have repeatedly seen protocols that used secure primitives but were insecure as a whole. Examples include the CRIME [RD12] and BREACH [GHP13] attacks against TLS, and the KRACK [VP17] attack against WPA2.

In light of this it is important that not only the security of our primitives is taken seriously, but that the entire protocol that makes use of them is

not insecure despite their security. The traditional approach of designing a scheme and deriving its security from the observation that nobody managed to break it has however turned out insufficient where larger protocols are concerned: Not only are there too many protocols and too few cryptanalysts to analyze all of them sufficiently, any vulnerability that is discovered after the publication would require all users to fix and update their implementations and even a lack of known successful attacks cannot prove their non-existence. Luckily, there is a better alternative in the form of security proofs.

1.3. Security Proofs

The most important practical countermeasure to insecure protocols that are built on secure primitives are security proofs that show the absence of vulnerabilities on the protocol level. Such proofs have limitations and can contain errors that invalidate their statements, but have the advantages that they can be completed before a protocol is deployed, do not have to hope for sufficient interest by cryptanalysis who may still overlook vulnerabilities and can even help as a tool to discover issues that aren't initially obvious.

It is important to understand though, that they cannot usually provide absolute proof of security, but have to rely on assumptions about the security of the used primitives.

Unconditionally proving the security of an encryption or signature scheme (other than versions of the one-time-pad) is widely accepted as infeasible, due to the close relationship to the P≠NP-assumption[2] which has turned out to be so hard to prove, that it is sometimes referred to as the "Holy Grail of Computer Science" and one of the seven Millennium Prize Problems by the Clay Mathematics Institute, whose solutions carry a reward of one million US-Dollars each.

As a substitute one might hope that it would be possible to at least prove the security of these schemes given the assumption that $P \neq NP$. Even this appears unlikely to be possible however [Imp95, AGGM06].

In light of all of this the best approach so far appears to be to largely use traditional cryptanalytic methods to analyze primitives and analyze protocols under the assumption that these primitives are secure.

[2] Strictly speaking the more relevant problem here is the BQP≠AM-assumption, but since $P \subseteq BQP \subseteq NP \subseteq AM$ and since there are good reasons to believe that $NP = AM$, this is unlikely to make things meaningfully easier.

1.3.1. Impact of Quantum Computers

Generally speaking, quantum computers do not impact the validity of the proof-based approach, but they can cause issues for certain techniques: In so far as proofs in the so called standard model are concerned, this primarily affects a technique known as rewinding, which involves copying the adversary's state before providing it with different values.

Furthermore, it complicates the use of the so called Random-Oracle Model (ROM) [BR93], which is a very common heuristic in that the traditional approach to use it does not work with queries in a quantum superposition, which led to the introduction of the Quantum Random-Oracle Model (QROM) [BDF+11] that can achieve similar things but is much more complicated to use.

Both of these techniques are primarily used in the analysis of primitives, whereas it is often sufficient for the analysis of communication-protocols to stay in the standard-model without rewinding, which applies to all proofs in this work.

1.4. Security Notions

In order to prove the security of a scheme, we of course need a formal definition of what the word "secure" means. Sadly, this question rarely has a simple answer, and intuitive approaches often turn out inadequate in practice:

One might for example be tempted to define the security of an encryption scheme along the lines of it having to be impossible to extract the plaintext from a ciphertext without the secret key. Such a definition would however allow for the leakage of very significant parts of the plaintext and fail to consider greater capabilities of the attacker, such as access to plaintext-ciphertext-pairs.

While this example may appear straightforward, the problem it presents is extremely widespread and can be incredibly subtle. In their attempts to acquire a high degree of adequacy many modern models have furthermore reached very high levels of complexity that not only make them inaccessible to laypeople, but in some instances even to experts. Considering that it is security models that define what the meaning of "secure" is in a given context and that the user of a system should ideally understand the security implications of using the system, this presents a practical issue for end users who can no longer understand intuitively what they can expect from a piece of software.

Sadly, there is so far not only no convincing solution, it isn't even clear how such a solution could look like, making this one of the hardest problems in cryptography.

On the positive side of things it bears mentioning however that there are many well established security notions for common primitives that seem to work well in practice; furthermore we find that formal definitions can reduce ambiguity for whether a given attack on a protocol should be considered to break the protocol, allowing for a more rigorous assessment.

1.5. Overview of this Thesis

As the first major part of this thesis we will introduce epochal signatures, a primitive that acts as a drop-in replacement for signatures but ensures deniability after a certain amount of real-world time has passed. This allows their use in settings that desire plausible deniability, such as certain chat platforms and to make the decision whether a given protocol should provide that property based on social and political factors, rather than based on technical limitation. Furthermore we introduce the (to our knowledge) first formalization of deniability in the setting of group-chats and prove that a large class of protocols, including Messaging Layer Security (MLS) [BBR+23] can be made to meet a fairly strong notion of deniability if the signatures used by them get replaced by epochal signatures.

After this we introduce Post-Quantum WireGuard, a modified handshake for the WireGuard Virtual Private Network (VPN), that fully eschews pre-quantum crypto and instead uses a modified version of Saber (then still a promising candidate in the NIST competition, before it lost to Kyber) and Classic McEliece to achieve highly efficient post-quantum security. We furthermore modified an existing proof for classical WireGuard to match Post-Quantum WireGuard and thereby showed that the new handshake achieves confidentiality and authenticity against quantum adversaries that are comparable to what the classical handshake achieves against classical adversaries.

We then took this approach further and created Post-Quantum Noise, where we introduced KEM-based key-exchanges to the Noise framework. This effectively meant the introduction of a wide variety of post-quantum handshake protocols that cover most use cases. We furthermore wrote a generic security proof for all PQNoise handshakes, which is a feat that has so far not even been achieved for classical Noise.

Lastly, we designed a key update protocol for satellite communication that is based on a modified PQNoise handshake. Several factors related to the

use in satellite communication created an unusual setting. Most notably the protocol was not meant to compute a session key for a directly following messaging phase, but rather a relatively long-term symmetric key for general communication that may only sporadically be replaced. Furthermore, the known position of satellites combined with the very narrow communication channels used between them and ground open an interesting possibility to skip some authentication.

2. Background

In this chapter we will introduce some of the primitives that we will repeatedly use throughout this work, as well as some simple lemmas that will repeatedly be applicable.

2.1. Notation

We will avoid implicit multiplication in formulas, aka xy will refer to a variable with a name that has two letters, rather than to the product of x and y. If we really mean the product, we will write $x \cdot y$.

We may refer to protocol specific values or functions x that appear for a large class of protocols, either as x if we use them in general or the protocol in question is clear from context, or for a specific protocol Y as $Y.x$ to indicate that we mean the specific variant associated with that protocol.

"$\mathbb{B}$" will refer to a boolean value, that is $\mathbb{B} = \{0, 1\}$.

$\Pr[X|Y]$ refers to the probability of a statement X being true, given the context created by the statement(s) Y.

$\Pr[\text{break}_i]$ indicates the highest achievable adversarial advantage of game i in a game-hopping proof.

We will distinguish three types of parties in this work, related to their runtime and computational power. PPT (Probabilistic Polynomial Time) will refer to the set of parties that run in polynomial time, that is they are "efficient" and classical but have access to randomness. QPT (Quantum Polynomial Time) will refer to the set of parties that can perform quantum operations (and thus have access to randomness as a side effect) but still have to run in polynomial time. TM (Turing Machines) will refer to the set of unbounded parties that can run arbitrary (quantum or classical) algorithms without time limitations.

2.2. Primitives

Certain primitives will come up in most chapters of this work. Instead of defining them each time anew, we will provide detailed definitions here and refer to them whenever needed.

2.2.1. Non-Interactive Key Exchanges (NIKE)

Modern Authenticated Key Exchange protocols (AKEs) are built from basic cryptographic primitives. The most widely used primitive for this purpose is Diffie-Hellman Key Exchange (DHKX) [DH76], a two party, non-interactive key exchange (NIKE) protocol.

In a NIKE, both parties are in possession of a key pair consisting of a private key a (resp. b) and a public key A (resp.B). When party $\mathcal{A}$ receives the public key B she can derive a shared secret $k = \mathsf{NIKE.derive}(a, B)$ by applying the derive function to her private key a and the public key B. The same shared secret $k = \mathsf{NIKE.derive}(b, A)$ can be computed by $\mathcal{B}$ using his private key and $\mathcal{A}$'s public key. Note that this protocol does not require any interaction between the two parties if the public keys are known (e.g., were retrieved from a public server).

While NIKEs are a very versatile primitive, DHKX, which is the only one that is widely used in practice, is fully broken by Shor's algorithm if the adversary has access to a sufficiently powerful quantum computer. So far, there is in fact no post-quantum secure NIKE that is commonly considered secure. (First proposals that are widely considered to be too new for practical use, do exist though [CLM+18, DKS18, RS06, Cou06, GdKQ+23].)

The NIST post-quantum cryptography standardization process did not consider NIKEs.

Diffie-Hellman Key Exchange (DHKX)

The DHKX requires a publicly known generator g of a cyclic group $\mathbb{G}$ with order p for which we will henceforth use multiplicative notation.

The involved parties then sample secret key x/y from $\mathbb{Z}_p^\times$ and computes $X := g^x / Y := g^y$ as their respective public keys.

$\mathsf{DHKX.derive}$ then works by raising the peer's public key by one's own private key: $Y^x = g^{y \cdot x} = g^{x \cdot y} = X^y$.

We believe that for certain groups there is no efficient pre-quantum algorithm that can find $g^{x \cdot y}$ when only given g, g^x, and g^y. If this holds we say that the Computational Diffie-Hellman assumption (CDH for short) holds in

the group. This is often a sufficient assumption to build not just NIKEs, but a whole host of other primitives as well.

A stronger requirement that is sometimes preferred is however that $g^{x \cdot y}$ should not just be hard to find, but indistinguishable from a group-element that has been chosen uniformly at random. This assumption is widely known as the Decisional Diffie-Hellman assumption, or DDH for short. We believe that there is no efficient pre-quantum algorithm that can break this assumption in certain well-chosen groups, such as sufficiently large prime-order subgroups of $\mathbb{Z}_q^\times$, where q is prime.

That being said, the often complex ways in which DHKX is used in many protocols means that often even DDH is insufficient to prove the security of complex schemes. In these cases, we have to reach for the family of so called PRF-ODH assumptions ("Pseudo Random Function with Oracle Diffie-Hellman") [BFGJ17]. These assumptions make statements about a setting where the output of the DHKX is directly combined with a user-chosen message and only require that final output to be indistinguishable from random. The problem with most of these assumptions is that they are very strong ("daring") in that they give the adversary indirect access to values that are derived from the shared secret of the DHKX, which in many instances prevents reductions to standard assumptions about the DHKX and the pseudo random function (PRF).

2.2.2. Key Encapsulation Mechanisms

In place of NIKEs, NIST selected Key Encapsulation Mechanisms (KEMs) which are closely related to (and usually built from) public key encryption (PKE) schemes. A PKE takes the receiver's public key and a message as input and produces the ciphertext encrypting the message as output. The receiver decrypts the ciphertext with their private key to obtain the message. In practice, a PKE is most often used to encrypt a random message which is afterwards hashed to obtain a shared key to be used with symmetric cryptography. A KEM is in some sense a limitation of a PKE to this very use case. Hence, a KEM is merely used for key transport. A KEM takes as input only the receiver's public key and produces two outputs: the ciphertext and a shared key. Only the ciphertext is sent to the receiver who then uses their private key to decapsulate the ciphertext and obtain the same shared key.

Definition 1 (KEM)**.** Formally a Key-Encapsulation Mechanism (KEM) is a tuple of three algorithms [Den03]:

- Gen is a probabilistic algorithm that takes a security parameter 1^λ and returns a keypair $(pk, sk) \in \mathcal{PK}_\lambda \times \mathcal{SK}_\lambda$.

- Enc is a probabilistic algorithm that takes a public key $pk \in \mathcal{PK}_\lambda$ and returns a ciphertext c from a ciphertext-space $\mathcal{C}_\lambda$ and a shared secret ss from a secret-space $\mathcal{SS}_\lambda$.

- Dec takes a secret key $sk \in \mathcal{SK}_\lambda$ and a ciphertext $c \in \mathcal{C}_\lambda$ and returns a shared secret $ss \in \mathcal{SS}_\lambda$.

Ideally decapsulation always produces the same shared secret as encapsulation, but for many KEMs in NIST's PQC competition there is a small probability δ that this is not the case. This property is generally known as *δ-correctness* and stands in contrast to the *correctness* in the more traditional sense where $\delta = 0$. In line with distinguishing between perfect, statistical, and computational security in the context of other security-notions, we argue that the case of $\delta = 0$ should best be referred to as *perfect correctness* in situations in which the distinction matters.

Definition 2 (Correctness)**.** We say that a KEM K is δ-correct if:

$$\forall \lambda \in \mathbb{N} : \Pr\left[\mathsf{Dec}(sk, c) \neq k \;\middle|\; \begin{array}{c} pk, sk := \mathsf{Gen}(1^\lambda) \\ c, k := \mathsf{Enc}(pk) \end{array}\right] < \delta$$

We say that K is perfectly correct if $\delta = 0$.

The probability here is specifically that of a specific individual encapsulation for a given, honestly generated public key producing inconsistent shared secrets.

The standard security notion for KEMs is $\mathsf{IND\text{-}CCA}$, which is short for "INDistinguishability under Chosen Ciphertext Attacks". Indistinguishability here refers to the goal that no efficient adversary should have a non-negligible advantage over guessing for distinguishing the key encapsulated in a ciphertext c from a random and independent one. The second component of this notion is then the setting in which the adversary has to attempt this distinction: For Chosen Ciphertext Attack, this means that he gets access to the public key and a decapsulation oracle for which he may freely choose what ciphertexts to request decapsulations for, as long as they are not the challenge-ciphertext.

Definition 3 (IND-CCA)**.** We say that a KEM K offers INDistinguishability under Chosen Ciphertext Attacks (IND-CCA) if:

$$\begin{aligned}&\forall \mathcal{A} \in \mathrm{QPT}, \lambda \in \mathbb{N} : \Pr\left[\mathsf{Exp}^{\mathsf{IND\text{-}CCA}}_{K,\mathcal{A},q}\left(1^\lambda\right) = 1\right] - \frac{1}{2} \\ &=: \mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{K,\mathcal{A},q}\left(1^\lambda\right) \leq \mathrm{negl}\left(\lambda\right),\end{aligned}$$

where $\mathsf{Exp}^{\mathsf{IND\text{-}CCA}}_{K,\mathcal{A},q}$ is defined as in Experiment 1.

Here, q is the maximum number of oracle-queries that the adversary may perform. Most of the time we will leave it out, in which case it can be thought of as being set to ∞, which means that the number of decapsulation queries is only bounded by the runtime of the adversary.

Experiment 1: $\mathsf{Exp}^{\mathsf{IND\text{-}CCA}}_{K,\mathcal{A},q}$, the IND-CCA-experiment for a KEM K.

$pk, sk := K.\mathsf{Gen}(1^\lambda)$
$c^*, k_0 := K.\mathsf{Enc}(pk)$
$queries := 0$
$k_1 \leftarrow_\$ \mathcal{SS}_\lambda$
$b \leftarrow_\$ \mathbb{B}$
Oracle $\mathsf{Dec}(c)$**:**
    $queries += 1$
    abort if $(queries > q \vee c = c^*)$
    return $K.\mathsf{Dec}(sk, c)$
$b' := \mathcal{A}^{\mathsf{Dec}}(pk, c^*, k_b)$
return $b = b'$

In some settings we will also use the weaker notion of IND-CPA, or "INDistinguishability under Chosen Plaintext Attacks", which differs in that the adversary does not receive access to the decapsulation oracle. Intuitively speaking IND-CCA is commonly associated with security against active adversaries who can interact with honest parties in malicious way, whereas IND-CPA provides security against adversaries that are passive and have to break the scheme only with access to the public key and the challenge-ciphertext.

Definition 4 (IND-CPA)**.** We say that a KEM K offers INDistinguishability under Chosen Plaintext Attacks (IND-CPA) if:

$$\begin{aligned}&\forall \mathcal{A} \in \mathrm{QPT}, \lambda \in \mathbb{N} : \Pr\left[\mathsf{Exp}^{\mathsf{IND\text{-}CPA}}_{K,\mathcal{A}}\left(1^\lambda\right) = 1\right] - \frac{1}{2} \\ &=: \mathsf{Adv}^{\mathsf{IND\text{-}CPA}}_{K,\mathcal{A}}\left(1^\lambda\right) \leq \mathrm{negl}\left(\lambda\right),\end{aligned}$$

where $\mathsf{Exp}^{\mathsf{IND\text{-}CPA}}_{K,\mathcal{A}}$ is defined as in Experiment 2.

Experiment 2: $\mathsf{Exp}^{\mathsf{IND\text{-}CPA}}_{K,\mathcal{A}}$, the IND-CPA experiment for a KEM K.

$pk, sk := K.\mathsf{Gen}(1^\lambda)$
$c^*, k_0 := K.\mathsf{Enc}(pk)$
$k_1 \leftarrow_\$ \mathcal{SS}_\lambda$
$b \leftarrow_\$ \mathbb{B}$
$b' := \mathcal{A}(pk, c^*, k_b)$
return $b = b'$

In an asymmetric setting these names are occasionally found to be slightly confusing, but it can help with intuition to consider their symmetric counterparts, where there is no public key that the adversary can use to create his own ciphertexts: Instead the symmetric IND-CPA-game has to provide an encryption oracle, for which the adversary can then *choose* the *plaintext* for which he will receive a ciphertext. Accordingly, in IND-CCA-games the adversary *chooses* the *ciphertext* that the decryption oracle will decrypt. The asymmetric case then only differs in that the encryption-oracle is replaced with access to the public key.

Lastly we note for completeness the existence of a more rarely used notion known as IND-CCA-1 which is a weakened version of IND-CCA in which the adversary runs in two phases, where it receives the public key and the decryption oracle in the first phase but only acquires the challenge ciphertext in the second phase where it no longer has access to the decryption oracle. In situations in which this notion is used, IND-CCA is sometimes referred to as IND-CCA-2 to distinguish them. Because we don't use that notion in this work, we will refrain from defining it from formally defining it here.

2.2.3. Signatures

While KEMs are an incredibly versatile tool that can not only be used to establish confidentiality, but also authenticity via so called challenge-response protocols, these have to be interactive and are not transferable, that is a transcript is not convincing for non-involved parties. This is where signatures come in. Like KEMs they are asymmetric primitives with a private and a public key, where the former can be used to sign any document and the latter to verify the resulting signature.

Definition 5 (Signature Scheme)**.** A signature scheme Σ is a tuple of three algorithms:

- $\Sigma.\mathsf{Gen}$, which takes the security parameter 1^λ, with $\lambda \in \mathbb{N}$ as argument and returns a public key $pk \in \mathcal{PK}$ and a secret key $sk \in \mathcal{SK}$;
- $\Sigma.\mathsf{Sign}$, which takes a secret key $sk \in \mathcal{SK}$ and a message m from the message-space $\mathcal{M}$ as arguments and returns "the" signature σ;
- $\Sigma.\mathsf{Verify}$, which takes the public key $pk \in \mathcal{PK}$, the signature σ and a message $m \in \mathcal{M}$ as arguments and returns "1" if the signature is accepted and "0" otherwise.

The message-space $\mathcal{M}$ is usually the set of arbitrary bitstrings of polynomial length in the security parameter, but there are also schemes in which this set is sometimes severely more restricted.

We say that a signature scheme is complete, if the verification algorithm accepts all honestly generated signature-message pairs. Or more formally:

Definition 6 (Completeness of Signatures)**.** We say that a signature scheme Σ is complete if:

$$\lambda \in \mathbb{N}, m \in \Sigma.\mathbb{M} :$$
$$\Pr\left[\Sigma.\mathsf{Verify}(pk, \sigma, m) = 1 \middle| \begin{matrix} pk, sk := \Sigma.\mathsf{Gen}(1^\lambda) \\ \sigma := \Sigma.\mathsf{Sign}(sk, m) \end{matrix}\right] = 1,$$

where $\Sigma.\mathbb{M}$ is the message-space of Σ.

Like with KEMs, there exist multiple notions of unforgeability for signatures, but the one that is used by far the most and has established itself as the de-facto standard is Existential UnForgeability under Chosen Message

Attacks or EUF-CMA [GMR88] for short. Intuitively it requires that no efficient attacker should have any meaningful chance to create a valid signature for any fresh message, even when given access to an unrestricted signing oracle. ("Fresh" means that signatures for messages for which she has received a valid signature from the oracle don't count as valid forgeries.) Or more formally:

Definition 7 (EUF-CMA for Signatures)**.** We say that a signature scheme Σ offers Existential UnForgeability under Chosen Message Attacks (EUF-CMA) if:

$$\begin{aligned}&\forall \mathcal{A} \in \mathrm{QPT}, \lambda \in \mathbb{N} : \Pr\left[\mathsf{Exp}^{\mathsf{EUF\text{-}CMA}}_{\Sigma,\mathcal{A}}\left(1^\lambda\right) = 1\right] \\ &=: \mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\Sigma,\mathcal{A}}\left(1^\lambda\right) \leq \mathrm{negl}\left(\lambda\right),\end{aligned}$$

where $\mathsf{Exp}^{\mathsf{EUF\text{-}CMA}}_{\Sigma,\mathcal{A}}$ is defined as in Experiment 3.

Experiment 3: $\mathsf{Exp}^{\mathsf{EUF\text{-}CMA}}_{\Sigma,\mathcal{A}}$, the EUF-CMA experiment for a signature scheme Σ.

```
pk, sk := Σ.Gen(1^λ)
M := {}
Oracle Sign(m):
    M ∪= {m}
    return Σ.Sign(sk, m)
σ*, m* := A^Sign(pk)
return Σ.Verify(pk, σ*, m*) ∧ m* ∉ M
```

2.2.4. Authenticated Encryption [with Associated Data]

While asymmetric cryptography is what enables modern secure communication as a whole, it is often too inefficient to protect the large payload of many modern applications. This is where traditional symmetric cryptography, that uses the same key for the sender and the receiver, comes in: Efficient building blocks like AES enable the creation of highly secure and very efficient encryption schemes that not only protect the confidentiality of their plaintexts but even allow protection of their authenticity and integrity as well as the authenticity and integrity of associated data.

This class of encryption schemes is widely known as Authenticated Encryption with Associated Data or "AEAD" for short and was first introduced by Rogaway [Rog02].

Definition 8 (AEAD)**.** An AEAD-scheme is a tuple of three algorithms:

- Gen is a probabilistic algorithm that takes a security parameter 1^λ, with $\lambda \in \mathbb{N}$ and returns a key k from a keyspace $\mathcal{K}$.
- Enc takes a key $k \in \mathcal{K}$, a nonce n from a nonce-space $\mathcal{N}$, a message m and associated data ad, both from a message-space $\mathcal{M}$ and returns a ciphertext c from a ciphertext-space $\mathcal{C}$.
- Dec takes a key $k \in \mathcal{K}$, a nonce $n \in \mathcal{N}$, a ciphertext $c \in \mathcal{C}$ and associated data $ad \in \mathcal{M}$ and returns a message $m \in \mathcal{M}$ or an error $\perp$.

Definition 9 (AEAD-correctness)**.** An AEAD-scheme AE is perfectly correct if:

$$\forall k \in \mathcal{K}, n \in \mathcal{N}, m, ad \in \mathcal{M} : AE.\mathsf{Dec}(k, n, AE.\mathsf{Enc}(k, n, m, ad), ad) = m$$

Definition 10 (AEAD-confidentiality)**.** An AEAD-scheme AE is $\mathsf{IND\$\text{-}CPA}$-secure if:

$$\begin{aligned}
&\forall \mathcal{A} \in \mathrm{QPT}, \lambda \in \mathbb{N} : \\
&\left(\begin{array}{c}\Pr\left[\mathcal{A}^{AE.\mathsf{Enc}(k,\cdot,\cdot,\cdot)}(1^\lambda) = 1 \middle| k \leftarrow AE.\mathsf{Gen}(1^\lambda)\right] \\ -\Pr\left[\mathcal{A}^{\$(\cdot,\cdot,\cdot)}(1^\lambda) = 1\right]\end{array}\right) \\
&=: \mathsf{Adv}^{\mathsf{IND\$\text{-}CPA}}_{AE,\mathcal{A}}(1^\lambda) \leq \mathrm{negl}(\lambda)
\end{aligned}$$

where $\mathcal{A}$ is not allowed to query the oracle with the same nonce twice and $\$$ is an oracle that returns random ciphertexts of the same length as a call to $AE.\mathsf{Enc}$ would return on all queries.

We say that $\mathcal{A}$ wins the game, if it outputs 1 when given access to the real oracle and if it outputs a value different from one, when given access to the random oracle.

Definition 11 (AEAD-authenticity)**.** An AEAD-scheme AE offers authenticity if:

$$\begin{aligned}
&\forall \mathcal{A} \in \mathrm{QPT}, \lambda \in \mathbb{N} : \Pr\left[\mathsf{Exp}^{\mathsf{auth\text{-}aead}}_{AE,\mathcal{A}}(1^\lambda) = 1\right] \\
&=: \mathsf{Adv}^{\mathsf{auth\text{-}aead}}_{AE,\mathcal{A}}(1^\lambda) \leq \mathrm{negl}(\lambda),
\end{aligned}$$

where $\mathsf{Exp}^{\mathsf{auth\text{-}aead}}_{AE,\mathcal{A},b}$ is defined as in Experiment 4.

Experiment 4: $\mathsf{Exp}^{\mathsf{auth\text{-}aead}}_{AE,\mathcal{A}}$, the security experiment for an AEAD-scheme AE.

$k := AE.\mathsf{Gen}(1^\lambda)$
$N := \emptyset$
$C := \emptyset$
Oracle $\mathsf{Enc}(n, ad, m)$**:**
  abort if $(n \in N)$
  $c := AE.\mathsf{Enc}(k, n, m, ad)$
  $N \cup = \{n\}$
  $C \cup = \{(n, c, ad)\}$
  return c
$n, c, ad := \mathcal{A}^{\mathsf{Enc}}(1^\lambda)$
abort if $((n, c, ad) \in C)$
return $AE.\mathsf{Dec}(k, n, c, ad) \neq \perp$

2.2.5. Pseudorandom Functions (PRF)

Many protocols need to combine secrets with other values to derive random-appearing values. A common example of this is to combine shared secrets from KEMs with values that identify the involved parties or other shared secrets. The most common building-block for this are PseudoRandom Functions (PRFs):

Definition 12 (PRF)**.** A PseudoRandom Function (PRF) H is an algorithm that takes a key k from a keyspace $\mathcal{K}$ and a message m from a message-space $\mathcal{M}$ and returns a value *rand* from a space $\mathcal{R}$.

Usually we find that $\mathcal{K}$ and $\mathcal{R}$ are the spaces of bitstrings of a fixed width and that $\mathcal{M}$ is the space of arbitrarily long bitstrings.

Definition 13 (PRF-Security)**.** We say that a function H is a secure Pseudorandom Function (PRF) if:

$$\forall \mathcal{A} \in \mathrm{QPT}, \lambda \in \mathbb{N} :$$
$$\Pr\left[\mathsf{Exp}^{\mathsf{PRF}}_{\mathsf{H},\mathcal{A}}(1^\lambda) = 1\right] - \frac{1}{2} =: \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{H},\mathcal{A}}(1^\lambda) \leq \mathrm{negl}(\lambda),$$

where $\mathsf{Exp}^{\mathsf{PRF}}_{\mathsf{H},\mathcal{A}}$ is defined as in Experiment 5.

We say that $\mathcal{A}$ wins the game, if it correctly outputs the challenge-bit b and looses otherwise.

Experiment 5: $\mathsf{Exp}^{\mathsf{PRF}}_{\mathsf{H},\mathcal{A}}$, the security experiment for a PRF H.

$b \leftarrow_{\$} \mathbb{B}$
$k \leftarrow_{\$} \mathcal{K}$
$queries := \emptyset$
Oracle $\mathsf{H}'(m)$**:**
  abort if $(m \in queries)$
  $queries \cup = \{m\}$
  if $b = 0$**:**
    return $\mathsf{H}(k, m)$
  else:
    $r \leftarrow_{\$} \mathcal{R}$
    return r
return $\mathcal{A}^{\mathsf{H}'}(1^{\lambda}) = b$

Especially when the PRF is used to combine multiple secrets, the roles of the arguments may change; in this case we need a definition with swapped arguments:

Definition 14 (PRF-SWAP)**.** We say that a function H is a secure swapped Pseudorandom Function (PRF-SWAP) if it is a secure PRF when its arguments are swapped.

Often we may in fact require that a function is both a secure PRF and a secure swapped PRF, to deal with different situations. In this case we call it a dual-PRF:

Definition 15 (dual-PRF)**.** We say that a function H is a secure dual Pseudorandom Function (dual-PRF) if it is both a secure PRF and a secure PRF-SWAP.

2.2.6. PseudoRandom Permutations (PRP)

A pseudorandom permutation (also known as "blockcipher") can be thought of as a PRF, where the message-space and the image-space are identical and function is a bijection, with an inverse function that is efficiently computable given the key. It is primarily useful as a building-block for other primitives.

Definition 16 (PRP)**.** A PseudoRandom Permutation is a tuple of three algorithms:

- Gen is a probabilistic algorithm that takes a security-parameter 1^λ and returns a secret key k.
- Enc is a deterministic algorithm that takes a secret key k and a bitstring m of a fixed length n and returns a bitstring c of the same length.
- Dec is a deterministic algorithm that takes a secret key k and a bitstring c of length n and returns a bitstring m of the same length.

Definition 17 (PRP-Completeness)**.** A PRP is perfectly complete if:

$$\forall \lambda \in \mathbb{N}, m \in \mathbb{B}^n : \\ \Pr\left[\mathsf{Dec}(k, \mathsf{Enc}(k, m)) = m \middle| k := \mathsf{Gen}(1^\lambda)\right] = 1$$

Definition 18 (IND-CPA)**.** We say that a PRP PRP offers INDistinguishability under Chosen Plaintext Attacks (IND-CPA) if and only if:

$$\forall \mathcal{A} \in \mathrm{QPT}, \lambda \in \mathbb{N} : \Pr\left[\mathsf{Exp}^{\mathsf{IND\text{-}CPA}}_{PRP,\mathcal{A}}(1^\lambda) = 1\right] - \frac{1}{2} \\ =: \mathsf{Adv}^{\mathsf{IND\text{-}CPA}}_{PRP,\mathcal{A}}(1^\lambda) \leq \mathrm{negl}(\lambda),$$

where $\mathsf{Exp}^{\mathsf{IND\text{-}CPA}}_{PRP,\mathcal{A}}$ is defined as in Experiment 6.

Experiment 6: $\mathsf{Exp}^{\mathsf{IND\text{-}CPA}}_{PRP,\mathcal{A}}$, the security experiment for an IND-CPA-PRP PRP.

```
M := dict()
k := PRP.gen(1^λ)
b ←$ 𝔹
Oracle Enc′ (m):
    if b = 0:
        return Enc(k, m)
    else:
        if m ∉ M:
            M[m] ←$ 𝔹^n
        return M[m]
b′ := 𝒜^{Enc′}(1^λ)
return b = b′
```

2.2.7. Message Authentication Codes (MACs)

A Message Authentication Code is a primitive that associates a message with knowledge of a secret key and thereby acts like a symmetric signature.

Definition 19 (Message Authentication Code (MAC))**.** A MAC is a tuple of three algorithms:

- Gen which takes a security-parameter 1^λ, with $\lambda \in \mathbb{N}$ and generates a secret key $sk \in \mathcal{K}$.
- auth which takes the key $k \in \mathcal{K}$ and a message $m \in \mathcal{M}$ and produces an authentication tag τ ("the" MAC), and
- Verify which takes the key $k \in \mathcal{K}$, a message $m \in \mathcal{M}$ and a tag and returns 1 if the tag is valid for the key and the message and 0 otherwise.

Usually Gen simply samples a random bitstring and Verify just runs auth again and compares the result to the tag, but neither of these is required to be the case, the latter may for example not be the case if auth is non-deterministic.

The standard security notion for MACs is Existential UnForgeability under Chosen Message Attacks or EUF-CMA for short. Intuitively it says that it should be infeasible for an attacker $\mathcal{A}$ to produce a MAC for any message for which she has not already seen a valid MAC, even if she is given access to an auth-oracle. Or more formally:

Definition 20 (EUF-CMA for MACs)**.** We say that a Message Authentication Code MAC offers Existential UnForgeability under Chosen Message Attacks (EUF-CMA) if:

$$\forall \mathcal{A} \in \mathrm{QPT}, \lambda \in \mathbb{N} : \Pr\left[\mathsf{Exp}^{\mathsf{EUF\text{-}CMA}}_{MAC,\mathcal{A}}(1^\lambda) = 1\right] =: \mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{MAC,\mathcal{A}}(1^\lambda) \leq \mathrm{negl}(\lambda),$$

where $\mathsf{Exp}^{\mathsf{EUF\text{-}CMA}}_{MAC,\mathcal{A}}$ is defined as in Experiment 7.

For simple MACs, where Gen just samples random bits and Verify just recomputes and compares the authentication-tag, we may also use MAC as a shorthand for MAC.auth.

2.2.8. Collision-Resistant Hash Functions

The last common primitive that we will regularly need is that of a hash function, which as a first intuition can be thought of as a function that will output a (seemingly) random value depending only on the input it receives.

Experiment 7: $\mathsf{Exp}^{\mathsf{EUF\text{-}CMA}}_{MAC,\mathcal{A}}$, the unforgeability game for a message authentication code MAC.

$k := MAC.\mathsf{Gen}(1^\lambda)$
$queries = \emptyset$
Oracle $\mathsf{auth}(m)$**:**
  $queries \cup = \{m\}$
  return $MAC.\mathsf{auth}(k, m)$
$m^*, \sigma := \mathcal{A}^{\mathsf{auth}}(1^\lambda)$
abort if $(m^* \in queries)$
return $MAC.\mathsf{Verify}(k, \sigma, m^*)$

Definition 21 (Hash-function)**.** Formally a hash-function is a function that takes one argument m from its message-space $\mathcal{M}$ and returns a value from its image $\mathcal{I}$.

For the purposes of this work, we require that $\mathcal{M}$ is the set of all (arbitrarily long) bitstrings and that $\mathcal{I}$ is the set of bitstrings of specific size.

There are many properties that such functions can have and that are relevant in cryptography, but the one we need the most is that of collision-resistance: The property that it should be infeasible to find two inputs that get mapped to the same output.

A common issue with formally defining this notion is the technicality that the mere existence of a collision implies an efficient attacker with non-negligible success-probability, namely the algorithm that just outputs the collision. This is of course without practical relevance, as one still has to find such a collision to be able to state this attacker, but properly formalizing the idea that there should be no "easy to find" attacker is non-trivial. In light of this the conventional approach is to treat collision resistance not as a property of an individual function, but as a property of a family of functions, from which a random function is sampled. This solves the issue since formally the hash-function is sampled after the attacker is already fixed, which means that any attacker that outputs a fixed collision would have to output one that is valid for a non-negligible subset of the function-family.

Definition 22 (Collision Resistance)**.** We say that a family $\mathbf{H}$ of functions is collision resistant for a security-parameter $\lambda \in \mathbb{N}$ if:

$$\forall \mathcal{A} \in \mathrm{QPT} : \Pr\left[\mathsf{Exp}^{\mathsf{Coll\text{-}Res}}_{\mathbf{H},\mathcal{A}}(1^\lambda) = 1\right] =: \mathsf{Adv}^{\mathsf{Coll\text{-}Res}}_{\mathbf{H},\mathcal{A}}(1^\lambda) \leq \mathrm{negl}(\lambda),$$

where $\mathsf{Exp}^{\mathsf{Coll\text{-}Res}}_{\mathbf{H},\mathcal{A}}$ is defined as in Experiment 8.

Experiment 8: $\mathsf{Exp}^{\mathsf{Coll\text{-}Res}}_{\mathbf{H},\mathcal{A}}$, the collision-resistance experiment for a family of hash-functions $\mathbf{H}$.

$H \leftarrow_{\$} \mathbf{H}$
$m_0, m_1 \leftarrow \mathcal{A}(H, 1^\lambda)$
return $m_0 \neq m_1 \wedge H(m_0) = H(m_1)$

The way we then use this definition in practice is that we treat the concrete hash-function that a specific protocol uses as though it had been sampled from a collision-resistant family, though we usually don't make that explicit.

2.3. Indistinguishability of Correct Ciphertexts

For the most part the δ-correctness (instead of perfect correctness) of KEMs affects the correctness of the protocols that are using them, which is usually considered acceptable, because δ tends to be smaller than the probability of events such as package loss over the internet or even cosmic radiation flipping bits. However δ-correctness also has a small effect on the soundness of the protocols: In the standard Indistinguishability-games that are used to define soundness for KEMs (e.g IND-CCA and IND-CPA) the adversary receives a challenge-ciphertext and has to distinguish the encapsulated key from randomness. Assume now that the adversary managed to learn the secret key of the KEM. Normally this would trivially break Indistinguishability, because the adversary could just decapsulate the challenge-ciphertext. In a δ-correct scheme however it is possible (with probability at most δ) for the challenge-ciphertext to be "bad" and that decapsulating it produces a result that is different from what the encapsulator (aka the challenger) received and there is no guarantee that the two are even related. Because of this a key-extraction attack on a δ-correct KEM does not directly imply a break of Indistinguishability.

To deal with this we introduce the notion of *Indistinguishability of Correct Ciphertexts* as a slight variation of the regular indistinguishability-game to which we will then relate it. For the most part we do this to simplify proofs and to avoid repetition when dealing with these cases.

Intuitively ICC works exactly like IND, except that the challenger decapsulates the challenge-ciphertext and causes an automatic adversarial win in the event that the encapsulated and the decapsulated keys mismatch.

Definition 23 (ICC-CCA)**.** We say that a KEM K offers Indistinguishability of Correct Ciphertexts under Chosen Ciphertext Attacks (ICC-CCA) if:

$$\begin{aligned}&\forall \mathcal{A} \in \mathrm{QPT}, \lambda \in \mathbb{N} : \Pr\left[\mathsf{Exp}^{\mathsf{ICC\text{-}CCA}}_{K,\mathcal{A}}(1^\lambda) = 1\right] - \frac{1}{2} \\ &=: \mathsf{Adv}^{\mathsf{ICC\text{-}CCA}}_{K,\mathcal{A}}(1^\lambda) \leq \mathrm{negl}(\lambda),\end{aligned}$$

where $\mathsf{Exp}^{\mathsf{ICC\text{-}CCA}}_{K,\mathcal{A}}$ is defined as in Experiment 9.

Experiment 9: $\mathsf{Exp}^{\mathsf{ICC\text{-}CCA}}_{K,\mathcal{A}}$

```
pk, sk := K.Gen(1^λ)
c*, k_0 := K.Enc(pk)
if k_0 ≠ K.Dec(c*):
    return 1
k_1 ←$ 𝒦
b ←$ 𝔹
Oracle Dec(c):
    abort if (c = c*)
    return K.Dec(sk, c)
b' := 𝒜^Dec(pk, c*, k_b)
return b = b'
```

Lemma 1. *For all security parameters λ, all adversaries $\mathcal{A}$, and all δ-correct KEMs K:*

$$\mathsf{Adv}^{\mathit{IND\text{-}CCA}}_{K,\mathcal{A}}(1^\lambda) \leq \mathsf{Adv}^{\mathit{ICC\text{-}CCA}}_{K,\mathcal{A}}(1^\lambda) + K.\delta$$

Proof. (ICC-CCA) To demonstrate the claim we will use game-hopping.

Let ***Game 0*** be the regular ICC-CCA-game. Then we find by definition that:

$$\Pr[\mathrm{break}_0] = \mathsf{Adv}^{\mathsf{ICC\text{-}CCA}}_{K,\mathcal{A}}(1^\lambda)$$

In ***Game 1*** we modify the game, such that we no longer check whether the challenge-ciphertext will be decapsulated correctly by the secret key.

This changes the potentially observable behavior in all cases where that check would normally fail. But because the challenge-ciphertext is generated (honestly) by the challenger and because there is only one challenge-ciphertext, the probability of running into that case is by definition at most

$K.\delta$. As a consequence the probability that any adversary can distinguish *Game 0* and *Game 1* is upper bounded by this event happening and therefore:

$$\Pr\left[\text{break}_1\right] \leq \Pr\left[\text{break}_0\right] + K.\delta$$

At this point we can note that *Game 1* is in fact already the IND-CCA game for K and therefore find:

$$\mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{K,\mathcal{A}}(1^\lambda) = \Pr\left[\text{break}_1\right] \leq \mathsf{Adv}^{\mathsf{ICC\text{-}CCA}}_{K,\mathcal{A}}(1^\lambda) + K.\delta$$

□

All of this applies equally in the case of adversaries without access to a decryption-oracle (CPA-adversaries):

Definition 24 (ICC-CPA)**.** We say that a KEM K offers Indistinguishability of Correct Ciphertexts under Chosen Plaintext Attacks (ICC-CPA) if:

$$\begin{aligned}&\forall \mathcal{A} \in \mathrm{QPT}, \lambda \in \mathbb{N} : \Pr\left[\mathsf{Exp}^{\mathsf{ICC\text{-}CPA}}_{K,\mathcal{A}}(1^\lambda) = 1\right] - \frac{1}{2}\\ &=: \mathsf{Adv}^{\mathsf{ICC\text{-}CPA}}_{K,\mathcal{A}}(1^\lambda) \leq \mathrm{negl}(\lambda),\end{aligned}$$

where $\mathsf{Exp}^{\mathsf{ICC\text{-}CPA}}_{K,\mathcal{A}}$ is defined as in Experiment 10.

Experiment 10: $\mathsf{Exp}^{\mathsf{ICC\text{-}CPA}}_{K,\mathcal{A}}$

$pk, sk := K.\mathsf{Gen}(1^\lambda)$
$c^*, k_0 := K.\mathsf{Enc}(pk)$
if $k_0 \neq K.\mathsf{Dec}(c^*)$**:**
  return 1
$k_1 \leftarrow_\$ \mathcal{SS}_\lambda$
$b \leftarrow_\$ \mathbb{B}$
$b' := \mathcal{A}(pk, c^*, k_b)$
return $b = b'$

Lemma 2. *For all security parameters λ, all adversaries $\mathcal{A}$, and all δ-correct KEMs K:*

$$\mathsf{Adv}^{\mathit{IND\text{-}CPA}}_{K,\mathcal{A}}(1^\lambda) \leq \mathsf{Adv}^{\mathit{ICC\text{-}CPA}}_{K,\mathcal{A}}(1^\lambda) + K.\delta$$

Proof. This follows directly from the proof of Lemma 1, by leaving out the pass-through for the decapsulation oracle, whose non-existence is the only difference between the experiments. □

3. Epochal Signatures

This chapter is with the exception of some reordering for practical intents identical to the paper "Epochal Signatures for Deniable Group Chats" [HW20], authored jointly with Andreas Hülsing, a slightly shortened version of which was published at IEEE S&P 2021 [HW21].

3.1. Introduction

In this work we take a formal look at deniability in group chat applications and introduce a new primitive that allows to turn many secure group chat protocols into deniable ones.

Deniability is a property of in-person conversation: As long as nobody is recording a conversation it is generally not possible to prove to someone not part of the conversation what was said (or even if the conversation happened at all). Recording a conversation without the consent of the other person (or approval of a judge) is usually considered immoral in most societies and even illegal in many jurisdictions.

Nowadays, we are moving large parts of our social conversations online. In this setting we are facing a dilemma. Communication tools that provide the traditional security properties of confidentiality, integrity, and authentication, often also provide a transferable proof of authenticity for messages or metadata that proves participation in the conversation. For example, modern tools for messaging in large groups, like MLS [BBR+23], sign every message to provide authentication in the presence of malicious insiders. This renders communication undeniable and changes things in a social setting: Suddenly, it is not one word against the other, but there is undeniable proof. Even more, it is now sufficient to leak a transcript which allows everyone to verify what was being said without the leaking party coming forward. In group communication this might even allow to make somewhat anonymous accusations. While this might be intended in some settings, it is obviously not intended in many others as we do not record most of our real-life conversations and would be suspicious if someone else would do this. There is a long debate that one can have about the settings in which deniability might be

useful and the degree of usefulness; see for example the discussion about deniability in MLS [tMp]. Instead of discussing this question, our work focuses on the technical aspects of deniability and its technical feasibility. Hence, in the following deniability is to be understood as technical deniability if not explicitly stated otherwise.

Probably the first work on deniability is "Undeniable Signatures" [CV90] by Chaum and van Antwerpen where they introduce signatures that can only be verified by a chosen recipient. This idea was further developed as chameleon signatures [KR98]. More explicitly, deniability of secure messaging appeared in the work on "deniable encryption" [CDNO97] which considered the deniability of encrypted messages even if the random coins, and possibly the secret key get corrupted.

The notion of deniability gained new relevance in the context of secure chat protocols. It first reappeared as plausible deniability in the proposal of the "Off-the-Record" protocol [BGB04] and was kept as an important privacy property of secure chat ever since [UDB+15a, tOt]. Consequently, the Signal-protocol [Mar13] that is now widely deployed in chat-software such as Signal and WhatsApp provides (some form of) deniability. The generic approach for two-party chat protocols to achieve deniability is to use a deniable key-exchange to setup a shared secret key. That shared secret key is then used to encrypt messages with an authenticated encryption scheme. Since everyone who can verify the authenticity of the ciphertext has the key necessary to create it, a ciphertext cannot serve as proof. While it is still not known how to achieve extremely strong notions of deniability efficiently [DKSW09], the above approach provides a practical solution for deniability of two-party chats.

A similar solution does not exist for group-chats: Using authenticated encryption with a shared group secret for symmetric authentication is insecure for the same reason for which it is deniable in the two-party case: All parties that have the key could have created a message, but since there are multiple parties, it could have come from any of them. In addition to that deniable key-exchanges don't necessarily scale well with more than two parties: Using a pairwise-approach works but requires a quadratic number of key exchanges in the number of parties and a linear number of authenticated encryptions per message.

Because of this most protocols either sacrifice deniability over authenticity (for example MLS [BBR+23]) or implement groups as essentially pairwise two-party-chats [LVH13, SVH18] which is inefficient in terms of communication complexity. Consequently, many protocols share further downsides, such as message-sizes linear in the group-size [SH19, Mar14] or the requirement that

there is at least one universally trusted user in every group [BST07]. An at least partial exception to this is "Multi-Party OTR" (mpOTR) [GUVC09] which uses a shared secret key for confidentiality but ephemeral signature keys for authenticity. This leads to an efficient protocol for message exchange, but the setup phase still has quadratic complexity in the group size. While the setup cost alone might be acceptable, there is a major downside to mpOTR. Any join or leave of a party requires the setup of a new chat. Given that for example MLS has the "aim to scale to groups as large as 50,000 members" [OBR+20], considering frequent join and leave operations, this renders mpOTR largely impractical for this scenario. A further issue is that deniability is only guaranteed after a chat has ended which also requires to frequently re-initialize chats even without join or leave operations.

Furthermore, we note that most of the existing work on deniability in chat protocols does not use formal models with generally agreed upon formal security notions but rather stay on an informal level. A notable exception is the work on online deniability [DKSW09] but the model is limited to two parties. Hence, existing schemes often just argue via the intractability of specific attacks [UG18] in place of a formal security argument.

In summary, there neither exists any satisfying solution for deniability in group chat nor a formal model that describes what deniability in a group chat setting actually means.

Despite its very different goal, "Efficient Post-Compromise Security Beyond One Group" [CHK19] is probably the closest to our work. In it the authors formalize signatures that "heal" after a compromise and note that this also allows some basic deniability. What they are proposing is quite similar to the signature-chains that we present in Section 3.10.2, but does not extend beyond that due to the different focus of the work and for example does not provide mechanisms to derive all expired all keys from any signature.

3.1.1. Our Contribution

In this work we solve the above problem. First, we introduce a formal framework for offline deniability in group-chats (to our knowledge the first one). For this, we build on the recent model for chats by Rösler, Mainka and Schwenk [RMS18] and extend it (see Sections 3.3 and 3.5). Our notion is parameterized by a predicate that states at what point in the execution deniability is achieved. Based on different such predicates we introduce three notions of different strengths for offline deniability. We show that our notions form a strict hierarchy and argue that the intermediate notion is the best choice for practical applications. We argue that our strongest notion,

which asks for immediate deniability of messages, is likely not achievable by practical protocols and discuss attack scenarios not covered by our weakest notion.

Second, we introduce the concept of *epochal signatures* which can be used to easily convert many existing, non-deniable protocols for group chat into deniable ones (Section 3.4). Our solution scales well to large group sizes and the resulting protocols are almost as efficient as the original ones. The idea is somewhat similar to mpOTR but avoids the requirement of pairwise exchange of temporary signature keys. Instead, *epochal signatures* evolve over time and allow for efficient forgeries of old messages after a fixed period of time. Essentially, they are the opposite of forward-secure signatures: Publishing a secret key allows to forge signatures valid in the past but not in the future. We remark here that these resolve a previously brought-up issue [Rob] with just publishing old keys, namely that the owner of said keys might be unable to do so. Readers that are not interested in the formal modelling should be able skip forward to this part without too many issues.

Third, we present an efficient generic construction for epochal signature schemes whose security we prove secure in the standard model (Section 3.6). Our construction relies only on well-established primitives, namely forward secure signatures, pseudorandom functions, and timelock puzzles. Since our proof works in the standard-model and avoids problematic techniques like rewinding, an instantiation with post-quantum schemes would immediately give us post-quantum epochal signatures. While unintended, it seems that our epochal signature proposal also constitutes an instantiation for a Time-Forge Signature, a recently [SPG19] proposed primitive for deniable E-Mail in the presence of DKIM. Another potential use-case might be the use within OpenPGP in settings in which it can be guaranteed or at least expected that mails arrive within the validity period.

In Section 3.7 we demonstrate that epochal signatures can be used to convert a large class of non-deniable group-chats into deniable group-chats by just using them as a drop-in-replacement. Lastly, we outline a few more techniques that we believe to be potentially useful for deniability in group-chats but are too specialized to be usable for a generic drop-in replacement (Section 3.10).

3.2. Security Model for Chats

In this section we will begin by describing the general dimensions of deniability, before defining the general framework within which we will describe our

formal definitions. This framework is defined on top of a previous formalization of group-chats by Rössler, Mainka, and Schwenk [RMS18] that did not include deniability among its security goals.

3.2.1. Deniability

Deniability is the ability of a party Alice ($\mathcal{A}$) to plausibly deny towards a judge Judy ($\mathcal{J}$) that she sent a certain message or participated in a certain group, even if she did do so. We model deniability as $\mathcal{J}$ not being able to distinguish between the transcript of a real interaction and a simulated transcript that was created by a simulator Simon ($\mathcal{S}$). If $\mathcal{J}$ is incapable of distinguishing between the two cases, then $\mathcal{A}$ can plausibly claim that a transcript is not real. In our definitions we focus on the technical aspects of deniability and ignore real life factors like trust in the entity providing a transcript.

Of course, deniability can trivially be achieved by sacrificing authenticity: If no party has any identifying long-term secrets, then every transcript could have been generated by everyone. However, authenticity is critical in practice: If $\mathcal{A}$ thinks that she is talking to a certain party $\mathcal{B}$ but is really talking to $\mathcal{J}$ the outcomes may be very unfortunate to say the least. Hence, the challenge is not to achieve "deniability" but "deniability *despite* authenticity". This dichotomy is what makes designing deniable chat-protocols hard in the first place.

Deniability is a very generic term which allows for different definitions depending on the treatment of certain aspects. First of all, one can consider judges with different capabilities. The most basic distinction here is the amount of work that the judge can perform. The typical options here are unbounded judges and asymptotically efficient judges, that is judges that are limited to a runtime that is polynomial in a security-parameter with either classical ($\mathcal{J} \in \mathrm{PPT}$) or quantum computers ($\mathcal{J} \in \mathrm{QPT}$).

More specific to deniability is then the question of whether the judges are online: The traditional notion here are so-called offline judges who receive the transcript after the completion of the protocol execution. However, one may also consider online-judges who exchange messages with group-members while the communication is happening [DKSW09].

Next one has to decide whether parties should only be capable of denying that they sent specific messages or whether they can also deny that the interaction happened in the first place [UDB+15b]. We call these message-deniability and participation-deniability, the latter of which is strictly stronger.

Then there is the power of the simulator. Traditionally deniability means that everyone is capable of simulating transcripts but more restricted settings

in which for example only supposed participants are capable of simulating interactions are thinkable as well. We are however not aware of any previous work that treats this aspect explicitly instead of just targeting the former (stronger) notion, which we will henceforth call *universal deniability* as opposed to *non-universal deniability* as a catch-all term for everything that is weaker.

Lastly there is the question of corruption. That is how much secret information the judge receives, there are two major aspects to this, the first one being whether the judge receives (or even chooses) the long-term secrets of the involved parties. Given that unbounded judges can compute matching secret keys for every public key themselves, this distinction is a lot less relevant to them, compared to bounded judges.

The other aspect concerns state-reveals: Seized devices may contain session-states that a realistic judge would most likely accept as evidence and a strong deniability notion should ensure that this evidence is useless. Considering our model of chats, the strongest notion in this regard is to give the resulting session-states of all `exp` and `ch` actions to the judge; this may seem very strong, but reliably prevents most vulnerabilities with regards to session state reveals. Giving the session-states of the `ch` actions to the judge models the case where one or more traitors try to betray a user $\mathcal{U}$ and hand their devices to the Judge. We call the case where the judge chooses the keypairs of all parties and receives all intermediate session-states of `exp` and `ch`-actions *full corruption*. In this context we would also like to mention the current draft of OTRv4 [tOt] that even considers partial corruption of a party, meaning that $\mathcal{J}$ only learns parts of the traitor's secret. This is something that we don't consider here. In summary we consider the following dimensions:

- **adversarial model:** Whether the judge is unbounded, bounded but has a access to a quantum computer, or bounded without such access.
- **online judges vs. offline judges:** Whether the judge receives the transcript as it is generated or at some (to be defined) later point in time.
- **message-deniability vs. participation-deniability:** Whether only the messages of a communication or even the occurrence of the communication can be denied.
- **universal vs non-universal deniability:** Whether everyone is capable of simulating a transcript or just participants.

- **corruption:** The degree to which the judge learns the secrets of the involved parties. In the case of *full corruption* the judge learns all long-term secrets of all parties (for an unbounded judge this will rarely be an advantage however), as well as the session-states that succeed `exp` and `ch` actions.

We note that that spectrum is likely incomplete when looking at online-judges. It is for example possible to distinguish between adaptive and non-adaptive online judges, the former of which are capable of changing parties' behaviors on the fly, while the later are not. We only consider offline-deniability in this work however but note that formalizing online-deniability in a way that unifies the existing notions would certainly be desirable.

In general, we note that online judges seem to be mostly interesting with corruption. While the notion also works without, it essentially limits the judge to traffic-analysis; while that may be very dangerous, it does not give the judge a real advantage over offline-judges who receive a transcript of the network (given that the network-transcript will usually be from an independent party, there is little reason to believe that the judge would not trust it over testimonies). Online judges with corruption are in turn essentially shoulder-surfing a member of the chat. Depending on the degree of the corruption this may even mean that that corrupted party is running the chat on a device that is fully controlled by the judge, at which point it seems very doubtful to us that a real-world judge would consider any claim that the interaction was simulated a reasonable doubt, even if it were theoretically possible. We also note that very few protocols target online-deniability [UDB+15b].

3.2.2. Base Model

The model we use for group chats is a slight extension of that of Rösler, Mainka and Schwenk [RMS18]. To stay consistent with our own conventions we adjust the names of some variables slightly, but other than that and this paragraph this entire subsection is an almost verbatim copy of the original description:

The model assumes a central server, that receives messages from the respective senders, caches them, and forwards them as soon as the receivers are online. Hence the protocols are executed in an asynchronous environment in which only the server has to always be online.

Groups are defined as tuples

$$gr = (ID_{gr}, G_{gr}, G^*_{gr}, info_{gr}), G^*_{gr} \subseteq G_{gr} \subseteq \mathbf{U}$$

where $\mathbf{U}$ is the set of all users of the protocol, G_{gr} is the set of all members of the group gr, G^*_{gr} is the set of administrators of gr. The group is uniquely referenced by ID_{gr}. Additionally, a title and other usability information can be configured in $info_{gr}$.

We denote communicating users with uppercase letters in the calligraphic font $(\ldots, \mathcal{U}, \mathcal{V}, \cdots \in \mathbf{U})$ and administrators with an asterisk $(\mathcal{U}^* \in G^*_{gr})$ where relevant. Every user maintains long-term secrets for initial contact with other users and a session state for each group in which she is member. The session state contains housekeeping variables and secrets for the exclusive usage in the group. Messages delivered in a group are not stored in the session state. By distinguishing between delivery and receiving of messages, we want to emphasize that a received message is first processed by algorithms before the result is presented to the user.

In order to provide a precise security model for secure group instant messaging, we define a group instant messaging protocol as the tuple of algorithms

$$\Pi = ((\mathsf{snd}, \mathsf{rcv}), \\ (\mathsf{SndM}, \mathsf{Add}, \mathsf{Leave}, \mathsf{Rmv}, \mathsf{DelivM}, \mathsf{ModG}, \mathsf{Ack})).$$

The first two algorithms (snd, rcv) provide the application access to the network (network interface). Thereby snd outputs ciphertexts and rcv takes and processes ciphertexts. The latter seven algorithms process actions of the user or deliver remote actions of other users to the user's graphical interface (user interface). Each protocol specifies these algorithms and the interfaces among them. To denote that one algorithm algA has an interface to another algorithm algB we write $\mathsf{algB}^{\mathsf{algA}}$.

Every algorithm has modifying access to the session state of the calling party $\mathcal{U}$ for the communication in group gr.

- $\mathsf{snd} \rightarrow \vec{c}$: Sends a vector of ciphertexts to the central server.
- $\mathsf{rcv}^{\mathsf{snd},\mathsf{DelivM},\mathsf{ModG},\mathsf{Ack}}(c)$: Receives ciphertext c from the central server and processes it by invoking one of the delivery algorithms and possibly the snd algorithm.

Actions of user $\mathcal{U}$ are processed by the following algorithms, which then invoke the snd algorithm for distributing the actions' results to the members $\mathcal{V}_i \in G_{gr}$ of group gr:

- $\mathsf{SndM}(gr, m) \rightarrow id$: Processes the sending of content message m to group gr.

- $\mathsf{Add}(gr, \mathcal{V}) \rightarrow id$: Processes adding of user $\mathcal{V}$ to gr.
- $\mathsf{Leave}(gr)$ Processes leaving of user $\mathcal{U}$ from gr.
- $\mathsf{Rmv}(gr, \mathcal{V})$ Processes removal of user $\mathcal{V}$ from gr.

Every algorithm that processes the calling user's actions outputs a unique reference string id. Actions initiated by other users are first received as ciphertexts by the rcv algorithm and then passed to the following algorithms, which deliver the result to user $\mathcal{U}$:

- $\mathsf{DelivM} \rightarrow (id, gr, \mathcal{V}, m)$: Stores m with reference string id from sender $\mathcal{V}$ in group gr for displaying it to user $\mathcal{U}$.
- $\mathsf{ModG} \rightarrow (id, gr')$: Updates the description of group gr with $ID_{gr} = ID_{gr'}$ to gr' after the remote modification with reference string id.
- $\mathsf{Ack} \rightarrow id$: Acknowledges that action with id was delivered and processed by all its designated receivers.

3.2.3. Our extensions

We note that all members of a group can perform SndM and Leave, but only administrators can execute Add and Rmv. We refer to [RMS18] for security-definitions besides deniability. For deniability we extend the model in the following way. We denote the long-term secrets of a user $\mathcal{U}$ as $sk_{\mathcal{U}}$ (or just sk, if unambiguous), any publicly identifying information tied to $sk_{\mathcal{U}}$ as $pk_{\mathcal{U}}$. We call the tuple $(pk_{\mathcal{U}}, sk_{\mathcal{U}})$ $\mathcal{U}$'s key pair and denote her session state in a group gr as $ss_{\mathcal{U},gr}$.

We introduce the notion of the state of a protocol. Intuitively a state st is a full snapshot of an execution of a protocol as the challenger would maintain it; as such it contains the sets of the existing users $\mathbf{U}$ and groups $\mathbf{G}$, as well as all long-term key pairs and session states. We also introduce a partial state that does not contain the key pairs and session-states.

Definition 25 (State)**.** A *state* st of our protocol consists of:

- The set $\mathbf{U}$ of all users $\mathcal{U}$.
- The set $\mathbf{G}$ of all groups gr as defined previously.
- The long-term public keys PK of all users in $\mathbf{U}$.

- The long-term secret keys SK of all users in $\mathbf{U}$.
- The session states $ss_{\mathcal{U},gr}$ of all groups $gr \in \mathbf{G}$ and all users $\mathcal{U} \in gr$.

The tuple $(\mathbf{U}, \mathbf{G})$ forms the partial state $st.ps$ of a state st. Two states st_0, st_1 are equivalent($st_0 \equiv st_1$) if and only if their partial states are identical ($st_0.ps = st_1.ps$).

From here on we use the following convention: If we define that a value x consists of multiple values a, b, c, ..., then $x.a$ refers to the a-value of x. (E.g., $st.\mathbf{U}$ refers to the set $\mathbf{U}$ that is part of a particular state st.)

Given a state st, we also introduce the notion of a group-state st^{gr}, which contains all information regarding a particular group gr. Intuitively it contains all session states and long-term key pairs of any party in that group, as well as the group description. We also define the notion of a partial group-state, which only contains the group-description, however.

Definition 26 (Group State)**.** Let st be a state and gr be a group, then the *group state* st^{gr} consists of:

- The group gr.
- The long-term public keys PK of all users in gr.
- The long-term secret keys SK of all users in gr.
- The session state $ss_{\mathcal{U},gr}$ of gr of each user $\mathcal{U} \in gr$.

A *partial group state* $st^{gr}.ps$ consists only of gr.

For our formal notions of deniability judges have to decide whether a transcript is real or not. In order to allow them to choose interactions without having to provide them with full oracle-access, we introduce the notion of an instruction: An instruction is a tuple that tells the challenger to perform some action in the name of some user with some arguments and is marked with a type that indicates the circumstances under which the challenger shall execute the instruction. An instruction list is simply an ordered list of instructions.

Definition 27 (Instruction List)**.** An *instruction* i is a tuple that contains:

- A party $\mathcal{P}$,
- a user action $act \in \{\mathsf{SndM}, \mathsf{Add}, \mathsf{Leave}, \mathsf{Rmv}, \mathsf{rcv}\}$ (with the respective arguments),

- a timepoint *time* and
- a *type* $\in \{\texttt{exp}, \texttt{ch}, \texttt{ar}, \texttt{hid}\}$.

An *instruction list* *il* is an ordered list of instructions. For an instruction list *il* and $X \in \{\texttt{ch}, \texttt{ar}\}$ we use il_X to refer to the sublist of *il* that contains all tuples whose *type* is in $\{\texttt{exp}, \texttt{hid}, X\}$. We call these two sublists the *executable sublists*.

Intuitively the executable sublists represent the possible executions that the judge will have to distinguish as illustrated in Figure 3.1. The type-names `exp`, `ch`, `ar` and `hid` are shorthands for "exposed", "challenge", "alternate reality" and "hidden".

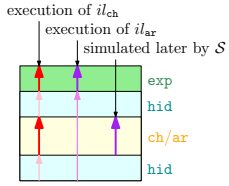

Figure 3.1.: Alternative transcripts based on the type of an instruction. The judge receives either an entirely real transcript (bold red arrows) or a partially simulated one (bold purple arrows).

In order to prevent trivial attacks, we furthermore introduce the following notion of consistency for instruction lists:

Definition 28 (Consistency)**.** An instruction list *il* is *consistent* with a starting state *st* if executing either of $il_{\texttt{ch}}$ and $il_{\texttt{ar}}$ with *st* as starting state

- is compliant with the protocol and
- all intermediate states that directly precede an `exp` action are equivalent between the executable sublists with regards to the target-group of that action.

We define the predicate is_consistent, which receives an instruction list *il* and a starting state *st*, to return 1 if and only if *il* and *st* are consistent.

Executing an instruction *inst* will (usually) cause messages to be sent over the network. The list of all these messages, each together with their sender

and receiver(s), as well as the resulting session state form an *instruction transcript*. Note in this context that a single user action, such as sndM may cause multiple executions of snd and rcv among different parties.

With this we define the *transcript* of an execution of an executable sublist of an instruction list as the concatenation of the instruction transcripts of all its actions. We include the session-states in the transcript to model corruption. While this may seem excessively powerful, we note that we mostly deal with unbounded judges in this work, who would usually be able to extract most of this information from the sent messages anyways; furthermore, we will present more specific rationales where appropriate.

Definition 29 (Transcript)**.** Let *instts* be the *instruction transcript* of executing an instruction $inst =: (\mathcal{P}, (act, args), time, type)$ in a state *st*. Then *instts* contains:

- For each snd-operation that is performed as part of executing *inst*:
  - The party $\mathcal{P}$ that performed snd.
  - The output $\vec{c}$ of snd.
- For each rcv-operation that is performed as part of executing *inst*:
  - The party $\mathcal{P}$ that performed rcv.
  - The ciphertext c that rcv receives.
- the group-state st^{gr} that results from executing *inst*.

The transcript *ts* of an execution of an executable sublist il_X of an instruction list *il* is the concatenation of the instruction transcripts of all instructions in il_X.

For the actual execution we define the algorithm exec that takes an executable sublist il_X of an instruction list *il* and a starting state *st* and then executes the sequence of instructions given by il_X, returning the resulting transcript and state.

Definition 30 (exec)**.** The algorithm exec takes a starting state *st*, and an executable sublist il_X of an instruction list *il* and executes all actions listed in *il* with *state* as starting state. It returns the complete transcript of the execution as well as the updated state of all involved parties. In the event that *il* orders an action that is not compliant with the protocol, an abort is raised.

We also define a version that only considers partial states, and does not return a transcript but only the resulting partial state:

Definition 31 (partial_exec)**.** The algorithm partial_exec takes a partial starting state ps and an executable sublist il_X of an instruction list il. It returns the partial state ps' that would result from executing il_X with any starting state whose partial state is ps. In the event that il_X orders an action that is not compliant with the protocol, an abort is raised.

Intuitively deniability means that every transcript could have been generated by everybody. We model this using a simulation-based definition, introducing a *simulator* $\mathcal{S}$ that has the goal to produce a *simulated transcript* that is indistinguishable from the real transcript of a group communication.

In order for $\mathcal{S}$ to be able to do its work, it needs to know what it has to simulate, so it has to receive some information about the instruction list il and the state st. Giving it the entire instruction list appears quite unrealistic to us however, which is why we introduce the notion of the *simulation instruction* $sim_{il,s}$ that contains only the information that $\mathcal{S}$ needs: Which instructions it has to simulate as well as the members of the group in which that instruction occurs:

Definition 32 (Simulation Instruction)**.** Let il be an instruction list and st be a state so that is_consistent $(il, st) = 1$. Then the *simulation instruction* $sim_{il,s}$ for il and st is an ordered list that contains:

- All entries $il[i]$ of $il_{\texttt{ch}}$ for which $il[i].type = \texttt{ch}$ and
- if there is no ch-entry for the group gr in which $il[i]$ is performed in $il_{\texttt{ch}}$ that directly precedes $il[i]$ in gr: $sim_{il,s}$ also contains the partial group state of gr before instruction $il[i]$. (That is the return-value of partial_exec $(il_{\texttt{ch}}[0 \dots, i-1],\ st)^{gr}$).

Here $il[i]$ refers to the i'th entry of il.

We note that our definitions also introduce notation: While deriving an executable sublist $il_{\texttt{ch}}$ from an instruction list il would *for example* strictly speaking require an explicit algorithm, we will for the remainder of the work just assume that if there is an instruction list il, then deriving $il_{\texttt{ch}}$ is possible and use it without introducing it as a variable first. The same holds for other notation that we used in this section, such as st^{gr} for group-states or simulation instructions $sim_{il,state}$.

3.3. Security Goals

There are many aspects to the security of group chats and our model "inherits" most of the more common aspects from the original model [RMS18]. Here we only outline authenticity (as it is directly relevant for our work) and introduce deniability. For all other notions we refer to [RMS18].

3.3.1. Authenticity

The most basic definition of authenticity is *message authentication* [RMS18]:

If a message m is delivered to $\mathcal{V} \in G_{gr}$ by $\mathsf{DelivM} \rightarrow (id, gr, \mathcal{U}, m)$, then it was indeed sent by user $\mathcal{U}$ by calling $\mathsf{SndM}(gr, m)$.

3.3.2. Naive Deniability

We can define a first notion of offline deniability that attempts to match the common intuition of what deniability is that we call *naive offline deniability*. In this case we allow the judge $\mathcal{J}$ to choose the instructions covered in the challenge protocol execution, with the exception of actions of type `exp` and `hid`. This models a case where a judge is given a transcript of a conversation and has to decide if this transcript is real or was prepared by a third party. In our model the judge is allowed to choose the key-pairs and the contents of the interaction in order to model the worst case. An equivalent but harder to work with definition would quantify over all possible key-pairs and transcripts. We point out that in this work we treat all adversaries in one game as using shared states.

Definition 33 (Naive Offline Deniability)**.** A protocol Π offers naive offline deniability or $\mathsf{N\text{-}OfD}$ if there is an efficient simulator $\mathcal{S} \in \mathrm{PPT}$ so that no judge $\mathcal{J}$ has a chance of winning $\mathsf{Exp}^{\mathsf{N\text{-}OfD}}_{\mathcal{S},\mathcal{J}}$, defined in Experiment 11 with a probability greater than $\frac{1}{2}$:

$\exists \mathcal{S} \in \mathrm{PPT} : \forall \mathcal{J} : \Pr[\mathsf{Exp}^{\mathsf{N\text{-}OfD}}_{\Pi,\mathcal{S},\mathcal{J}}(1^\lambda) = 1] \leq \frac{1}{2}$

(Many definitions require that the probability of a guessing game is $1/2$ or negligibly close to it. In our protocols we model aborts to cause whatever experiment is running to stop execution and return 0. This makes defining the adversarial advantage easier, but usually makes it trivial to create adversaries with a success-probability of zero by intentionally causing an abort, which is significantly different from $1/2$. We deal with this by requiring that the success-probability is less than or equal to the targeted $1/2$, which treats these adversaries as unsuccessful.)

Experiment 11: $\mathsf{Exp}^{\mathsf{N\text{-}OfD}}_{\Pi,\mathcal{S},\mathcal{J}}$. Naive offline deniability for chats.

```
1  b ←$ 𝔹
2  U, il, PK, SK := J()
3  st := (U, ∅, PK, SK, ∅)
4  abort if (¬is_consistent(il, st))
5  abort if (∃inst ∈ il : inst.type ∈ {hid, exp})
6  if b = 0:
7  |  _, ts := exec(il_ch, st)
8  else:
9  |  ts := S(PK, sim_{il,st})
10 b' := J(ts)
11 return b = b'
```

We call this notion "naive" because $\mathcal{J}$ does not receive access to *any* real transcript, even ones that are completely independent of the interaction in question. In particular this means that she won't even learn about public parameters of the server. Consider a protocol in which the server will sample a random bitstring once and attach it to any packet it sends from then on and where all honest parties simply ignore this bitstring. The naive deniability notion now allows $\mathcal{S}$ to sample this bitstring from the same distribution and use that sampled value in his simulation. Since $\mathcal{J}$ has no way to learn what bitstring is used in this model, $\mathcal{S}$ is able to get away with this technique. In the real world it would however be exceedingly unlikely, that $\mathcal{J}$ could not learn that bitstring by setting up an account herself and checking whether they match, implying that this notion would be too weak in practice for most purposes. The problem here is that this notion does not yet consider transcripts that are partially trusted by $\mathcal{J}$. Formally this manifests itself in the limitation of the challenges to those that only contain `ch-` and `ar`-actions (the latter of which are never used but are required for is_consistent).

A further issue, that is however relatively minor as $\mathcal{J}$ is unbounded, is that she does not receive any session-states as there are no `exp` actions, meaning that this model also only considers corruptions of keys instead of full corruption.

3.3.3. Strong Deniability

To protect against these kinds of attacks we consider the entire system as a whole. The straightforward demand here would be that the judge receives the real transcript of everything that happens, except for the interactions in question, which are either real as well or generated by a simulator. (We refrain from formalizing this here, as it will be a special case of our final notion.) While extremely powerful, this notion causes its own problems. Specifically, we conjecture that it is incompatible with all efficient protocols, unless other desirable features are sacrificed:

Conjecture 1. *There is no chat-protocol that offers post-compromise secrecy (also known as backward secrecy [UDB+15b]), and strong offline deniability without requiring a trusted party or an interaction with all members of a group after performing any non-*rcv *action in that group.*

We introduce this conjecture mostly to justify why we consider weaker notions desirable. Our rationale for it is as follows: At first consider any protocol that encrypts consecutive messages of a user with keys that are in a non-trivial relation to each other. Let now m_0 and m_1 be consecutive messages by the same user, so that they are encrypted with the keys k_0 and k_1 and let m_1 be part of the challenge that the judge $\mathcal{J}$ receives, but not m_0. This allows the following attack: If $\mathcal{J}$ is unbounded, she can extract k_0 and k_1 with at least high probability and check whether they are related. If the transcript is entirely real, then they will be related with probability 1. In contrast, if m_1 was encrypted by the simulator, this will likely not be the case, as the relation is by assumption non-trivial, and the simulator does not know k_0.

This attack works in particular for schemes that derive their keys from states that are derived from each other. Getting rid of this property is not without downsides: The first option would be to design a protocol that avoids ephemeral secret states entirely by only using long-term secrets. Such a protocol is trivially incapable of offering post-compromise secrecy. The second option would be to remove the possibility of linking consecutive states. This could be done by distributing ephemeral public keys amongst all possible senders and having them use each key only once. This leads to the same problem as before just with a shorter attack window, as well as the possibility of running out of fresh keys. We also are not aware of other methods that do this without interacting with all parties in the group – performing essentially a fresh handshake.

Note that this rationale does intentionally not make use of the session-states that are part of the transcript. We do this to show that the problem is not caused by our strong notion of corruption and cannot be circumvented by weakening that notion.

3.3.4. State Disassociation

Attacks like the one outlined in the previous section are a consequence of correlations between successive group-states. As such they become impossible, once two group-states become fully disassociated with each other. The precise condition of when and if that happens in a given group will be the main parameter to our generic model.

A formal definition of a state disassociation is strictly speaking not necessary for the definition of our model, but we expect that most proofs would end up defining some form of this notion as a stepping-stone anyways. Because of this we provide it here, with the hope that it will not only help with understanding the intentions behind the following sections, but also to reduce redundant work in proofs and the risk of multiple incompatible definitions.

Intuitively a *state disassociation* in a group gr is any sequence of actions that transforms a group-state st_0 into a group-state st_1 in such a way that it ensures that st_1^{gr} contains no information about st_0^{gr}.

A state disassociation predicate $\mathsf{sd\_pred}$ intuitively states if an instruction list achieves state disassociation for a given starting state. More precisely, it takes an instruction-list il, a partial starting-state ps and a group gr and returns true if the group state in gr that results from executing il on ps is uncorrelated to ps, and false otherwise. We formalize this, requiring that there is no judge $\mathcal{J}$ that can output two consistent pairs of starting states and instruction-lists which cause state-disassociations and result in equivalent (Definition 25) states, such that $\mathcal{J}$ can distinguish those states.

Definition 34 (State Disassociation Predicate)**.** A predicate $\mathsf{sd\_pred}$ is a state disassociation predicate if there is no adversary $\mathcal{J}$ that can win Experiment 12 with probability $> \frac{1}{2}$:

$\forall \mathcal{J} : \Pr[\mathsf{Exp}^{\mathsf{State\text{-}Disassoc}}_{\mathsf{sd\_pred},\mathcal{J}}(1^\lambda)] = 1] \leq \frac{1}{2}$

3.3.5. Disjoined Instruction Lists

To prevent the aforementioned attacks, it is not merely sufficient to define what a state disassociation is, but also the circumstances under which it has to occur. The goal in this regard is to cause a disassociation between all pairs

Experiment 12: $\mathsf{Exp}^{\mathsf{State\text{-}Disassoc}}_{\mathsf{sd\_pred},\mathcal{J}}$. State Disassociation Predicate

$st_0, il_0, st_1, il_1, gr, SK := \mathcal{J}(\mathsf{sd\_pred})$
for $b \in \mathbb{B}$**:**
  abort if $(\neg \mathsf{is\_consistent}(il_b, st_b))$
  abort if $(\neg \mathsf{sd\_pred}(il_b, st_b.ps, gr))$
  $\_, st'_b := \mathsf{exec}\left(st_b,\ il_{b_{\mathtt{ch}}}\right)$
abort if $\left(st_0'^{gr} \not\equiv st_1'^{gr}\right)$
$b \leftarrow_{\$} \mathbb{B}$
$b' := \mathcal{J}(st_b'^{gr}, st_{1-b}'^{gr})$
return $b = b'$

of longest consecutive sequences of actions in a group whose type is either only `exp` or only `ch` and all such sequences whose type is `ch` and the final resulting state.

For this we introduce the notion of *disjoined* instruction lists. It has a parameter sd_pred that has to be filled with a state-disassociation-predicate. With this we can now define that an instruction list il and a partial starting state ps are disjoined under a given state disassociation predicate sd_pred if:

For every group gr and every sublist il' of il that contains an action a in gr, where actions a', a'' in gr that directly precede/follow il have a different type from a and their types are in $\{\mathtt{ch}, \mathtt{exp}\}$, then il' satisfies the state-disassociation predicate for gr.

We remark that this bans partial changes to the states between `ch` and `exp` actions. If that ban was dropped, the simulator would require precise information about which states have to be updated in what way. While possible, this would vastly complicate the security notion, while likely not resulting in a stronger notion (if enough information is given to the simulator, it can simply simulate the `hid`-actions and remove them from the output). To give a more formal definition:

Definition 35 (Disjoined)**.** We say that an instruction list il and a partial starting state st are *disjoined* under a predicate sd_pred if the predicate disjoined, as defined in Algorithm 1, returns 1 when called with them.

Algorithm 1: Definition of disjoined for a state disassociation predicate sd_pred.

```
1  fun c_index(il, n):
2      return n − |{x ∈ il[0, ..., n] | x.type ≠ ar}|
3  fun disjoined_sd_pred(il, ps):
4      for i, j, k ∈ ℕ³, i < j < k < |il|:
5          t_i := il[i].type; t_j := il[j].type; t_k := il[k].type
6          g := il[i].group
7          i' := c_index(il, i); k' := c_index(il, k)
8          ps' := partial_exec(il_ch[0, ..., i'], ps)
9          if ¬sd_pred(il_ch[i', ..., k'], ps', g) ∧t_i, t_k ∈ {ch, exp} ∧t_j ∉ {t_i, t_k}:
10             return 0
11     return 1
```

3.3.6. Full Interaction

In order to provide a predicate that may work as a state disassociation predicate in many protocols while also being a plausible option for use in real-world protocols, we introduce the notion of a hidden full interaction. Intuitively a *full interaction* occurs if every member of a group performs an active action or is removed from the group. If a full interaction occurs in `hid` actions with no other types of actions in between, we call it a *hidden full interaction*; or more formally:

Definition 36 (Hidden Full Interaction)**.** Let il be a consistent instruction list as defined in Section 3.2 with starting state st. We say that il causes a hidden full interaction in a group $gr \in st.\mathbf{G}$, if there is a consecutive sublist il' in il, in which every action concerning gr is `hid` and for every party $\mathcal{P} \in gr$ at least one of the following holds:

- $\mathcal{P}$ successfully sends a (valid) message m:
  1. $\mathcal{P}$ executes $\mathsf{SndM}(gr, m)$ for any valid message m, getting the identifier id
  2. All parties $\mathcal{P}' \in gr, \mathcal{P}' \neq \mathcal{P}$ execute $\mathsf{rcv}(gr)$
  3. $\mathcal{P}$ successfully executes $\mathsf{Ack}(id)$
- $\mathcal{P}$ successfully leaves the group:
  1. $\mathcal{P}$ executes $\mathsf{Leave}(gr)$, getting the identifier id.

2. All parties $\mathcal{P}' \in gr, \mathcal{P}' \neq \mathcal{P}$ execute $\mathsf{rcv}(gr)$
3. $\mathcal{P}$ successfully executes $\mathsf{Ack}(id)$

- $\mathcal{P}$ is successfully removed from the group:
  1. An administrator $\mathcal{V}^*$ executes $\mathsf{Rmv}(gr, \mathcal{P})$, getting the identifier id.
  2. All parties $\mathcal{P}' \in gr, \mathcal{P}' \notin \{\mathcal{V}^*, \mathcal{P}\}$ execute $\mathsf{rcv}(gr)$
  3. $\mathcal{V}^*$ successfully executes $\mathsf{Ack}(id)$

We define the predicate HFI that takes an instruction list il, a starting state st and a group-identifier gr and returns 1 if and only if executing il with starting-state st causes a hidden full interaction in the group identified by gr.

Our suggestion for the state disassociation predicate is therefore that the predicate returns true if and only if a hidden full interaction occurs. We note that this mirrors the usual requirements for establishing post-compromise-secrecy, if we view pure key-updates as sending empty messages. We note that whether HFI is a secure state disassociation predicate remains a property of the protocol in question and has to be proven on a case-by-case basis.

3.3.7. Our Deniability Framework

In order to define the security notions that we actually recommend, we will use the framework depicted in Experiment 13. It follows the typical structure of a distinguishing game in which a judge $\mathcal{J}$ has to guess a randomly sampled bit b. The only ways for her to do this better than just random guessing are to extract information about the execution-history from the state and to distinguish whether the transcript of an interaction of her choice is either real ($b = 0$) or whether it was (partially) simulated by a simulator $\mathcal{S}$ ($b = 1$).

The experiment starts with an empty state. This does not really limit the power of $\mathcal{J}$, since she can always start il with `hid` actions that create a state whose partial state is whatever she likes. It would alternatively have been possible to let $\mathcal{J}$ output the starting state, but since she would then know it, all groups would have to perform a state-disassociation *before* executing `ch` or `ar` actions.

We allow $\mathcal{J}$ to pick the long-term key pairs of the involved parties, as we would give that information to her anyways in the end to deal with corruption and don't think that we should allow the existence of weak keys. The only requirements that we enforce for the operating instruction il that $\mathcal{J}$ outputs are that it is consistent with the empty starting state and properly disjoined under a state disassociation predicate $\mathsf{sd\_pred}$ that is left as model parameter.

Experiment 13: $\mathsf{Exp}^{\mathsf{OfD}}_{\Pi,\mathcal{S},\mathcal{J},\mathsf{sd\_pred}}$. The Experiment used to define our notions of offline deniability for protocols Π. The specific notion depends on the parameter sd_pred which specifies the way in which the operating instructions has to be disjoined.

$b \leftarrow_{\$} \mathbb{B}$
$\mathbf{P}, PK, SK, il := \mathcal{J}()$
$st := (\mathbf{P}, \emptyset, PK, SK, \emptyset)$
abort if $\left(\neg\mathsf{is\_consistent}\,(il,\ st) \vee \neg\mathsf{disjoined}_{\mathsf{sd\_pred}}\,(il,\ st)\right)$
if $b = 0$**:**
  $full\_transcript, st := \mathsf{exec}\,(st,\ il_{\mathtt{ch}})$
  $judged\_transcript :=$
    $(msg \in full\_transcript | msg.type \in \{\mathtt{exp}, \mathtt{ch}\})$
else:
  $full\_transcript, st := \mathsf{exec}\,(st,\ il_{\mathtt{ar}})$
  $simulated\_transcript := \mathcal{S}(PK, sim_{il,st})$
  $judged\_transcript :=$
    $\mathsf{merge}((msg \in full\_transcript | msg.type = \mathtt{exp})\,,$
    $simulated\_transcript)$
$b' := \mathcal{J}(judged\_transcript)$
return $b = b'$

Only giving the public keys and $sim_{il,st}$ to $\mathcal{S}$ means that all notions defined from this experiment provide universal deniability. This could be weakened to different forms of non-universal deniability by giving further information such as secret keys to $\mathcal{S}$, but as these notions are rarely if ever targeted in the literature and by no means standard, we refrain from doing so here.

We note that the transcript that $\mathcal{J}$ receives contains all session states that precede an exposed action in the respective group, which together with the adversarially chosen key pairs means that all notions defined in this model consider full corruption. We choose this notion not because we believe that it models anything particularly realistic, but because security against it implies security against many weaker forms of corruption and because it does not seem to cause any significant problems for protocol design compared to those weaker notions.

This entire experiment defines a family of security-notions that differ on the used state-disassociation predicate sd_pred. That predicate essentially defines at what point communication in a group becomes deniable. As such

the members of that family will vary substantially with the extreme cases being a predicate that always returns 1, requiring deniability after every instruction, and a predicate that always returns 0, meaning that no group provides deniability if there is ever an exp action in it. Any other predicate will provide something in between these two notions. As such sd_pred is a customization point that has a major effect on the practical deniability that a scheme provides and saying that a protocol provides offline deniability in the sense that no judge can win the above game better than by random guessing is a statement of very limited use without specifying sd_pred.

We remark that a state disassociation predicate p that outputs 1 strictly more often than another predicate p′, does not necessarily imply a stronger security notion: Consider the case where both predicates accept the same consistent instruction lists, but p also accepts all inconsistent ones, which are rejected by p′. The difference between p and p′ has no effect on the provided security-notion because the security-game performs a consistency-check anyways and aborts if it fails.

Because of the large effect that sd_pred has on our notion of OfD-security, we will provide three concrete instantiations of it, namely the extreme cases of *strong* and *weak* offline deniability, as well as a notion between the two, that is efficiently instantiable under reasonable requirements (compare Conjecture 1) while guaranteeing a much stronger form of deniability than the weak notion.

With this we will now define our notion of strong offline deniability:

Definition 37 (Strong Offline Deniability)**.** A protocol Π offers *strong offline deniability* or S-OfD if there is an efficient simulator $\mathcal{S} \in \text{PPT}$ so that no judge $\mathcal{J}$ has a chance of winning the OfD-game (Experiment 13), with sd_pred $= (x \mapsto 1)$ with a probability greater than $\frac{1}{2}$:

$$\exists \mathcal{S} \in \text{PPT} : \forall \mathcal{J} :$$
$$\Pr[\mathsf{Exp}^{\mathsf{OfD}}_{\Pi,\mathcal{S},\mathcal{J},(x\mapsto 1)}(1^\lambda) = 1] \leq \frac{1}{2}$$

This notion guarantees universal participation deniability under full corruption for offline-judges and is the strongest notion for offline deniability in chats that we consider in this work, as it fully covers all the security goals that we outlined for offline-deniability in subsection 3.2.1. We doubt however that it is *efficiently* achievable (Conjecture 1).

Because of this we also introduce a weaker notion, that we believe to be efficiently achievable in practice. Specifically, we suggest using the predicate HFI as defined in Definition 36. The reasoning behind this is that this notion still eventually achieves deniability in corrupted groups but does not require

an update of the entire group state after every operation. Instead, the state can be updated as a side-effect of regular messages, allowing for more practical protocols. Additionally, this notion has the advantage that it can be defined generically and therefore does not rely on any specifics of the protocol.

Definition 38 (HFI Offline Deniability)**.** A protocol Π offers *HFI offline deniability* or HFI-OfD if there is an efficient simulator $\mathcal{S} \in \text{PPT}$ so that no judge $\mathcal{J}$ has a chance of winning the OfD-game (Experiment 13), with $\mathsf{sd\_pred} = \mathsf{HFI}$ with a probability greater than $\frac{1}{2}$:

$$\exists \mathcal{S} \in \text{PPT} : \forall \mathcal{J} :$$
$$\Pr[\mathsf{Exp}^{\mathsf{OfD}}_{\Pi,\mathcal{S},\mathcal{J},\mathsf{HFI}}(1^\lambda) = 1] \leq \frac{1}{2}$$

Theorem 1. *S-OfD is strictly stronger than HFI-OfD.*

Proof. "S-OfD $\Rightarrow$ HFI-OfD": This follows directly from the fact that the only difference between the two notions is that the judge has strictly more freedom in choosing *il* in S-OfD. Therefore, any successful judge against a protocol Π in the HFI-OfD setting is also successful against Π in the S-OfD setting with exactly the same runtime and success-probability.

"HFI-OfD $\not\Rightarrow$ S-OfD": Our core idea here is to define a protocol in which every user $\mathcal{P}$ appends a random bitstring *bs* to successive messages in a group that does not change as long as only she sends messages (no change to group or messages by other users). In a fully real transcript, all values of *bs* of her consecutive messages are then equal, but since the simulator does not receive that value, he cannot simulate them in a consistent way.

Let Π be a HFI-OfD-secure chat-protocol. Then we define a protocol Π' which only differs from Π in that every party $\mathcal{P}$ samples a random bitstring *bs* before performing a SndM-action and appends it to all packages that she sends over the network for messages in the respective group. She resamples *bs* before her next message if and only if she is either the only member of the group, another party performs any active action or any party is removed from the group. All other parties ignore *bs*.

Π' is not S-OfD-secure since the judge $\mathcal{J}$ can output an operating instruction that creates this situation and in which a party sends an `exp` message, directly followed by a `ch`-message. Upon receiving the transcript, $\mathcal{J}$ can simply look at these two messages and check whether the bitstrings match. If $b = 0$ then the entire transcript is real and given the construction, the bitstrings match. If $b = 1$ the simulator has no way of knowing which bitstring

was used in the real execution and therefore has to guess; this guess will be wrong with a probability that is exponentially close to 1 in the length of the bitstring. Thus, any judge that outputs 0 if the bitstrings match and 1 otherwise, will win the distinction game with a probability greater than 1/2.

Π' is however still HFI-OfD-secure: If $\mathcal{P}$ is the only member in a given group, then Π' is constructed to not repeat bs, which means that $\mathcal{S}$ can sample it from the same distribution without problem. Otherwise, given the definitions of disjoined and HFI as well as the OfD game, any pair of messages in the operating instruction that $\mathcal{J}$ outputs in which one message is `exp` and the other is in {`ch`, `ar`} must have a hidden action by at least one other party between them; in that case $\mathcal{P}$ also samples a fresh and independent bs, meaning that it too can simply be simulated by $\mathcal{S}$ sampling it from the same distribution as $\mathcal{P}$. Because Π is HFI-OfD-secure all other parts of Π' can also be simulated by $\mathcal{S}$, meaning that Π' is HFI-OfD-secure.

Given this, the existence of a HFI-OfD-secure protocol implies the existence of a HFI-OfD-secure protocol that is not S-OfD-secure, therefore HFI-OfD-security does not imply S-OfD-security. □

Next, we define an even weaker notion for protocols that have trouble achieving HFI-OfD:

Definition 39 (Weak Offline Deniability)**.** A protocol Π offers *weak offline deniability* or W-OfD if there is an efficient simulator $\mathcal{S} \in$ PPT so that no judge $\mathcal{J}$ has a chance of winning the OfD-game (Experiment 13), with $\mathsf{sd\_pred} = (x \mapsto 0)$ with a probability greater than $\frac{1}{2}$:

$$\exists \mathcal{S} \in \text{PPT} : \forall \mathcal{J} :$$
$$\Pr[\mathsf{Exp}^{\mathsf{OfD}}_{\Pi,\mathcal{S},\mathcal{J},(x\mapsto 0)}(1^\lambda) = 1] \leq \frac{1}{2}$$

Intuitively W-OfD forces the judge to only create three kinds of groups: Fully corrupted ones in which all actions are `exp`, fully hidden ones in which all actions are `hid` and "target"-groups that contain only `ch`- and `ar`. This is because the definition of disjoined requires a state-disassociation between any pair of actions that don't fit into any of the above groups, but the definition of W-OfD-security means that there is no sequence of interaction that causes one.

We also note that the fully exposed groups will be of very little help for the judge in protocols in which sessions states are independent from each other except for the shared secret key (as long as that key is constant): Due to the independence `exp` groups are not affected by the challenge-bit b in any way

and thus don't contain any useful information about it. As such the judge has to judge the ch-groups only on the provided transcripts, which is why we consider the term "weak" justified, despite the seemingly strong form of corruption.

Theorem 2. *HFI-OfD is strictly stronger than W-OfD.*

Proof. "HFI-OfD $\Rightarrow$ W-OfD": This follows directly from the fact that the only difference between the two notions is that the judge has strictly more freedom in choosing *il*, therefore any valid W-OfD-attacker is also a valid HFI-OfD-attacker with exactly the same runtime and success-probability.

"W-OfD $\not\Rightarrow$ HFI-OfD": Our core idea here is to define a protocol in which every user $\mathcal{P}$ appends a random bitstring *bs* to all her messages in the same group. In a fully real transcript, all values of *bs* in the same group are then equal, but since the simulator does not receive that value, he cannot simulate it in a consistent way.

Let Π be a W-OfD-secure chat-protocol. Then we define a protocol Π' which only differs from Π in that every party $\mathcal{P}$ samples a random bitstring *bs* whenever she joins a group for the first time and appends it to all packages that she sends over the network for messages in that group. All other parties ignore *bs*.

Π' is not HFI-OfD-secure since the judge $\mathcal{J}$ can output an operating instruction in which $\mathcal{P}$ has to execute both an exp and a ch SndM-action in the same group, by separating it with a hidden full interaction. Upon receiving the transcript, $\mathcal{J}$ can simply look at these two messages and check whether the bitstrings match. If $b = 0$ then the entire transcript is real and given the construction, the bitstrings match. If $b = 1$ the simulator has no way of knowing which bitstring was used in the real execution and therefore has to guess; this guess will be wrong with a probability that is exponentially close to 1 in the length of the bitstring. Thus any judge that outputs 0 if the bitstrings match and 1 otherwise, will win the distinction game with a probability greater than $1/2$.

Π' is however still W-OfD-secure: Given the definitions of disjoined and the OfD game, any operating instruction that contains both ch- and exp-actions in the same group causes an abort. Therefore, any *bs* that $\mathcal{S}$ has to generate will be in a different group and independent of those that $\mathcal{P}$ generates. Because of this $\mathcal{S}$ can simply sample his bitstrings from the same distribution. Since $\mathcal{S}$ can also simulate the remaining parts of the protocol (since they are identical to Π), Π' is W-OfD-secure.

Given this, the existence of a W-OfD-secure protocol implies the existence of a W-OfD-secure protocol that is not HFI-OfD-secure, therefore W-OfD-security does not imply HFI-OfD-security. □

Corollary 2.1. *S-OfD is strictly stronger than W-OfD.*

Proof. This follows directly from the combination of Theorem 1 and Theorem 2. □

Theorem 3. *W-OfD is strictly stronger than naive offline deniability.*

Proof. "W-OfD ⇒ N-OfD ": We can convert every judge $\mathcal{J}$ that attacks N-OfD-security into a judge $\mathcal{J}$' that attacks W-OfD-security with the same success probability and essentially the same runtime. The judge $\mathcal{J}$' simply forwards all messages between $\mathcal{J}$ and the W-OfD game.

The instruction-list *il* that $\mathcal{J}$ outputs is disjoined because it only contains `ch` and `ar`-actions: Given the definition of disjoined, state-disassociations are only ever required if actions of two different types are performed in the same group. With *il* this is never the case; thus *il* is properly disjoined and accepted by the W-OfD-challenger.

Following that the challenge-transcripts *ts* are generated in exactly the same way and $\mathcal{J}'$ has to output the exact same value as $\mathcal{J}$. As such this translation between the games is perfect, has only a tiny (clearly polynomial) overhead and $\mathcal{J}'$ wins exactly when $\mathcal{J}$ is a successful judge in the weak offline deniability game. Therefore W-OfD-security implies N-OfD-security.

"N-OfD ⇏ W-OfD ": Our core idea here is to define a protocol in which every user $\mathcal{P}$ appends a random bitstring *bs* to all her messages in all groups. In a fully real transcript all values of *bs* are then equal, but since the simulator does not receive that value, he cannot simulate the groups with `ch` actions in a consistent way.

Let Π be an N-OfD-secure chat-protocol.

We define a protocol Π′ which only differs from Π in that every party $\mathcal{P}$ samples a random bitstring *bs* and appends it to the package of each SndM-action she performs. All other parties ignore *bs*.

Π′ is not W-OfD-secure since the judge $\mathcal{J}$ can output an operating instruction in which $\mathcal{P}$ has to execute both an `exp` and a `ch` SndM-action.

Upon receiving the transcript, $\mathcal{J}$ can simply look at any two such messages of the same user and check whether the bitstrings match. If $b = 0$ than the entire transcript is real and given the construction, the bitstrings match. If $b = 1$ the simulator has no way of knowing which bitstring was used in the real execution and therefore has to guess; this guess will be wrong with

a probability that is exponentially close to 1 in the length of the bitstring. Thus any judge that outputs 0 if the bitstrings match and 1 otherwise, will win the distinction game with a probability greater than $1/2$.

Π' is however still N-OfD-secure: Given the definition of N-OfD-security, the transcript that $\mathcal{J}$ receives is either entirely simulated or entirely real. Because of this $\mathcal{S}$ can simply sample bs from the same distribution as $\mathcal{P}$. Since $\mathcal{S}$ can also simulate the remaining parts of the protocol (since they are identical to Π) Π' is N-OfD-secure.

Given this, the existence of a W-OfD-secure protocol implies the existence of a N-OfD-secure protocol that is not W-OfD-secure, therefore N-OfD-security does not imply W-OfD-security. □

Given all of the above we conclude that while S-OfD is clearly the strongest notion, it will usually be too expensive to target in practical protocols. Because of this we recommend HFI-OfD as the target notion that will usually be desirable, as it is still quite strong but also efficiently achievable. W-OfD is a notion that is still weaker and that we consider to be the minimum that a protocol aiming at deniability should target outside of special circumstances.

We recommend against the use of N-OfD, despite the initial appeal it may have because of its simplicity: The assumptions it makes about judges are too optimistic for practical use outside of special circumstances.

3.4. Epochal Signatures

We now introduce signatures that become deniable after a certain amount of time but provide an unforgeability notion that is essentially equivalent to the standard notion of existential unforgeability under chosen message attacks (EUF-CMA) before that. These allow adding deniability to many efficient multi-party chats that use signatures for their authentication, such as MLS [BBR+23]) by simply replacing the used signature scheme;

3.4.1. Syntactic Definition

The main differences from a standard signature-scheme are the use of epochs and addition of a per-epoch public information $pinfo_e$. Knowing $pinfo_e$ should be enough to create arbitrary expired signatures; it must be made public in such a way that everyone has easy access to it. The reason for why we separate $pinfo_e$ from the signatures is simply so that parties don't have to have seen a real signature in order to create an expired one.

Definition 40. An epochal signature scheme Σ is a tuple of four algorithms: $\Sigma.\mathsf{gen}$, $\Sigma.\mathsf{evolve}$, $\Sigma.\mathsf{sign}$ and $\Sigma.\mathsf{verify}$.

- $\Sigma.\mathsf{gen}$: Takes a security-parameter 1^λ, an epoch-length Δt, the maximum number of epochs $E \in \mathsf{poly}(\lambda)$ and the number of epochs $V < E \in \mathbb{N}$ for which signatures are valid and returns a long-term key pair (pk, sk).
- $\Sigma.\mathsf{evolve}$: Takes the secret key sk and returns public epoch information $pinfo_e$ and an updated secret key sk' or $\perp$ if sk has already been evolved E times.
- $\Sigma.\mathsf{sign}$: Takes the secret-key and a message $m \in \mathbb{M}$ and returns a signature σ.
- $\Sigma.\mathsf{verify}$: Takes a public key pk, an epoch e, a signature σ and a message m and returns a boolean value b that tells whether the signature is valid in epoch e.

We leave the message-space $\mathbb{M}$ as a parameter, define the public/secret key space as the set of all possible values that $\Sigma.\mathsf{gen}$ can generate as first/second output, the signature space as the output-space of $\Sigma.\mathsf{sign}$ and the space of all public epoch information $pinfo_e$ as the output-space of $\Sigma.\mathsf{evolve}$.

An epochal signature scheme is complete if all honestly generated, unexpired signatures are accepted by the verification algorithm; or more formally:

Definition 41 (Completeness of Epochal Signatures)**.** An epochal signature Σ scheme is *complete* if:

$$\begin{aligned}
&\forall \lambda \in \mathbb{N},\ E \in \mathsf{poly}(\lambda),\ V \in \{1, \ldots, E-1\},\\
&pk, sk \leftarrow \Sigma.\mathsf{gen}\left(1^\lambda,\ \Delta t,\ E,\ V\right), e \in \{1, \ldots, E-1\},\\
&e' \in \{e, \ldots, \mathsf{min}\left(e+V,\ E\right)\} :\\
&\Sigma.\mathsf{verify}\left(pk,\ e',\ \Sigma.\mathsf{sign}\left(sk_e,\ m\right),\ m\right) = 1
\end{aligned}$$

Where sk_e is the secret key returned from the e'th execution of $\Sigma.\mathsf{evolve}$.

Since many of the following definitions will depend on the duration of operations and the specific points in time at which they are performed we introduce a utility function "now " that returns the wall-clock time at the point of execution at which it is invoked. We don't specify the format of the returned value, except that subtracting two returned values from each other has to produce the wall-clock duration that passed between their invocations and that they have to be less-than comparable.

3.4.2. Unforgeability

Intuitively our notion of unforgeability is the epoch-based equivalent of the standard notion of "Existential UnForgeability under Chosen Message Attacks" (EUF-CMA), the only difference being that signatures expire and are no longer accepted by the challenger afterwards. More Formally:

Definition 42 (Unforgability of Epochal Signatures (EEUF-CMA))**.** An epochal signature scheme Σ is unforgeable in the sense of *Epochal Existential UnForgeability under Chosen Message Attacks* or EEUF-CMA if there is no efficient forger $\mathcal{F}$ that has a non-negligible chance of winning Experiment 14:

$$\begin{aligned}&\forall \mathcal{F} \in \mathrm{QPT},\ \lambda \in \mathbb{N},\ E \in \mathsf{poly}(\lambda),\ V \in \{1, \dots, E-1\} : \\ &\Pr[\mathsf{Exp}^{\mathsf{EEUF\text{-}CMA}}_{\Sigma,\mathcal{F}}(1^\lambda, \Delta t, E, V)] = 1] \\ &=: \mathsf{Adv}^{\mathsf{EEUF\text{-}CMA}}_{\Sigma,E,V,\mathcal{F}}(1^\lambda, \Delta t) \leq \mathrm{negl}(\lambda)\end{aligned}$$

The restrictions on the points in time at which the adversary may perform which actions may appear unnecessary, as it would easily be possible to define schemes that just consider the epoch-counter. We add them anyways, as they allow among others the release of information encapsulated in time-lock puzzles to strengthen deniability, which would lead to trivial vulnerabilities without these restrictions. We note that while the requirement to check whether the signature in question is expired is not explicit in the game, it is implied by the way signatures are checked for freshness: Since only signatures created in the last V epochs are considered in the freshness-check, $\Sigma.\mathsf{verify}$ accepting any signature older than that could trivially be used to win the game, implying that the scheme in question does not provide EEUF-CMA-security.

We note that by prepending the epoch to the message and checking it during verification, every EUF-CMA secure scheme can be turned into a EEUF-CMA secure one. On the other hand, every EEUF-CMA secure scheme can be turned into an EUF-CMA secure by setting Δt to a very high value and only using the first epoch.

3.4.3. Deniability

We say that an epochal signature-scheme is deniable if there is a simulator $\mathcal{S}$ that can create arbitrary expired signatures that are indistinguishable from

Experiment 14: $\mathsf{Exp}^{\mathsf{EEUF\text{-}CMA}}_{\Sigma,\mathcal{F}}(1^\lambda, \Delta t, E, V)$. The unforgeability game for epochal signatures.

$pk, sk := \Sigma.\mathsf{gen}(1^\lambda,\ \Delta t,\ E,\ V)$
$t_0 := \mathsf{now}()$
$e := 0$
$queries := [\emptyset, \dots, \emptyset]$
fun $\Sigma.\mathsf{evolve}'()$**:**
  abort if $(t = E)$
  abort if $(\mathsf{now}() < t_0 + e \cdot \Delta t)$
  $e \mathrel{+}= 1$
  $pinfo_e, sk := \Sigma.\mathsf{evolve}(sk)$
  return $pinfo_e$
fun $\Sigma.\mathsf{sign}'(m)$**:**
  abort if $(\mathsf{now}() \geq t_0 + e \cdot \Delta t)$
  $\sigma := \Sigma.\mathsf{sign}(sk,\ m)$
  $queries[e] \cup= \{m\}$
  return σ
$\sigma, m := \mathcal{F}^{\Sigma.\mathsf{evolve}', \Sigma.\mathsf{sign}'}(pk)$
$ret := \Sigma.\mathsf{verify}(pk,\ e,\ \sigma,\ m)$
for $e' \in \{\mathsf{max}(0,\ e - V), \dots, e\}$**:**
  abort if $(m \in queries[e'])$
return ret

real ones. In order to do so $\mathcal{S}$ receives the public epoch information $pinfo_e$ of an epoch in which the simulated signature was already expired.

Definition 43 (Deniability of Epochal Signatures)**.** An epochal signature Σ scheme is *deniable* if there is an efficient simulator $\mathcal{S} \in \mathrm{PPT}$, so that no judge $\mathcal{J}$ can win Experiment 15 with a probability $> \frac{1}{2}$:

$$\forall \lambda \in \mathbb{N},\ E \in \mathsf{poly}(\lambda),\ V \in \{1, \dots, E-1\} : \exists \mathcal{S} \in \mathrm{PPT} :$$
$$\forall \mathcal{J} \in \mathrm{TM} : \Pr[\mathsf{Exp}^{\mathsf{Deniability}}_{\Sigma,\mathcal{S},\mathcal{J}}(1^\lambda, \Delta t, E, V)] = 1] \leq \frac{1}{2}$$

We give the secret key to $\mathcal{J}$ because we consider unbounded judges in the first place, and it should not help her in distinguishing signatures. This

Experiment 15: $\mathsf{Exp}^{\mathsf{Deniability}}_{\Sigma,\mathcal{S},\mathcal{J}}(1^\lambda, \Delta t, E, V)$. The deniability game for epochal signatures.

$pk, sk := \Sigma.\mathsf{gen}(1^\lambda,\ \Delta t,\ E,\ V)$
$b \leftarrow_\$ \mathbb{B}$
$m, e_0, e_1 := \mathcal{J}(pk, sk)$
$\sigma := \bot$
abort if $(e_0 + e_1 \geq E \vee e_0 < 0 \vee e_1 < V)$
for $e \in \{1, \ldots, e_0\}$**:**
  $pinfo_e, sk := \Sigma.\mathsf{evolve}(sk)$
if $b = 0$**:**
  $\sigma := \Sigma.\mathsf{sign}(sk,\ m)$
for $e \in \{e_0 + 1, \ldots, e_0 + e_1\}$**:**
  $pinfo_e, sk := \Sigma.\mathsf{evolve}(sk)$
if $b = 1$**:**
  $\sigma := \mathcal{S}(m, e, pinfo_{e_0+e_1})$
$b' := \mathcal{J}(\sigma, sk)$
return $b = b'$

essentially prevents the inclusion of information about previously generated signatures in the secret key, which we consider desirable.

An anonymous reviewer pointed out an out-of-model attack against this definition: If a party forwards an epochal signature before its expiration to a time-stamping server and receives a regular signature on it and the current time, then it is not possible to simulate that signature. In this case the time-stamping server acts as a witness that the signature was real. This is an out-of-model attack in both the signature-(as there are no time-stamping-oracles in the deniability game) and the chat-setting (Requirement 6 of Theorem 7) that cannot be prevented with any scheme whose deniability is based on delayed information-releases. Developing schemes that resist such attacks is an important challenge for future work.

3.5. Proposed Techniques

In this section we describe how we build an efficient epochal signature scheme that satisfies the security notions that we defined in the previous section. We do this by starting with a naive and inefficient scheme that we then modify.

This starting point is the scheme that simply layers two signatures on top of each other, where the lower "dynamic" one is replaced with each epoch updates, while the public key of the upper "static" one serves as long-term identity. Once the signatures created within an epoch expire, the secret key of the dynamic level is published and can be used to create expired signatures. We remark that this is similar to how CAs work, with the main difference that we publish expired secret keys intentionally. The resulting scheme works as follows:

Key-Generation is identical to the key-generation of the scheme used for the static layer, giving $(\widehat{pk}, \widehat{sk})$. At the start of an epoch e the signer generates a new dynamic key-pair $(\overline{pk}_e, \overline{sk}_e)$, signs $\overline{pk}_e$ and e with $\widehat{sk}$, giving $\hat{\sigma}_e$ and publishes $(e, \overline{pk}_e, \hat{\sigma}, [(\overline{sk}_0, \hat{\sigma}_0) \ldots, (\overline{sk}_{e-V}, \hat{\sigma}_{e-V}])$ as $pinfo_e$.

Signing is done by signing the message m with $\overline{sk}$, giving $\overline{\sigma}$ and outputting $(\overline{\sigma}, pinfo_e)$ as signature. Verification works by checking that $pinfo_e.e$ is less than V epochs in the past and verifying $\overline{\sigma}$ and $pinfo_e.\hat{\sigma}$. The simulation of expired signatures works by using the expired dynamic secret key and the signature on its public-key from $pinfo_e$ to recreate the signature in question.

3.5.1. Deterministic Bottom-Layer

The main problem with the above solution is the size of the public epoch information $pinfo_e$ which is caused by the need to include the dynamic secret-keys and signatures under the static key for all expired epochs. A simple method to remove the former is to deterministically generate them based on a seed that can be derived from the seeds of later epochs.

Assume that we want to use E epochs. Then during key-generation we sample a random bitstring r_E and apply a Pseudo Random Function (PRF) H on it E times, storing all intermediate values in the secret key. More precisely, we use r_e as the key to a pseudorandom function H that we call with an independent fixed value m as message $(\mathsf{H}(r_e, m))$ and use the resulting value as r_{e-1}. Whenever we use a probabilistic algorithm during the e'th key-evolution (most notably $\overline{\Sigma}.\mathsf{gen}\,(1^\lambda)$), we use $\mathsf{H}(r_e, m')$ instead of the randomness, where $m' \neq m$ is a different message from the one used for computing r_{e-1}. This way all dynamic secret keys can be removed from $pinfo_e$ by adding r_{e-V} to it, which drastically decreases its size. The main disadvantage of this method is that the size of the secret key becomes linear in E, which we will deal with further below by using a pebbling algorithm.

We note that this kind of repeated hashing is similar to Lamport's approach for repeated authentication with passwords [Lam81], though used differently in a different context.

3.5.2. Reversed Forward-Secure Signatures

While the previous subsection goes a long way in reducing the size of $pinfo_e$, that size is still linear in the number of expired epochs, due to the signatures under the static key for all past epochs. To solve this, we again use pebbling but this time with forward secure signatures. Forward secure signatures were introduced as an answer to the problem that if an attacker receives the key of a signature-scheme, he can forge arbitrary signatures and were first formalized by Bellare and Miner [BM99] in 1999. They add epochs and key-updates to regular signatures: Every signature is marked as having been created in a certain epoch; epoch-updates are performed by the signer by updating the secret key so that it can no longer be used to sign messages for previous epochs. In the case of a key-compromise the adversary can then only create valid signatures for the current and later epochs, but the signatures for previous epochs stay secure. The previously mentioned paper does not give an explicit name for its unforgeability notion, but it is colloquially known as "Forward Secure Existential UnForgeability under adaptive Chosen Message Attacks" or FS-EUF-CMA, which is what we will use henceforth.

Forward-secure signatures provide the guarantees that we want, except backwards in time: Instead of all signatures in the past remaining secure in the event of a compromise, we want all future ones to remain secure. We resolve this in the same way as in the previous subsection: Assume that we want to use E epochs. Then during key-generation we evolve the initial secret key of the forward-secure scheme E times and store all derived keys as secret keys of our epochal signature scheme. To sign a message for the i'th epoch with the epochal scheme, we then use the secret key of the forward-secure scheme that resulted from $(E-i)$ evolutions. Since we store the secret keys of all epochs of the forward-secure signature scheme, this works with constant time-complexity. With this we can then modify $pinfo_e$ to only include the $(E-(e-V))$'th secret key of the forward secure scheme, as the secret keys of all previous epochs can be derived from it via the key-evolution function of the forward secure scheme. Given that there are efficient, hash-based forward-secure-signatures with reasonably small state, such as XMSS [BDH11] we can even get efficient post-quantum-security.

3.5.3. Pebbling

The main problem with the previous two solutions is that they require either a very large secret key or a very high amount of computations per epoch-update (both linear in the number of remaining epochs). We solve this problem by using pebbling schemes which allow us to compute the same values in logarithmic time and space. For a detailed description of how they work we refer to the work of Schoenmakers [Sch16] and only note that they reduce the size of the secret key to be logarithmic in E at a low computational overhead.

Formally we will treat them as a pair of Algorithms PebblePrep and Pebble of which the former takes an initial value x_0, a function f and an integer n and returns a state s_{n-1}. The latter takes a state s_i and returns an updated state s_{i-1} and $f^i(x_0)$ for $i \in \{0, \dots, n-1\}$.

3.5.4. Undeniable Deniability

One of the issues with publishing the secret-keys once they are no longer needed is that doing so requires the ability to publish; if Alice loses internet-access before doing so and there are witnesses that this happened, she loses some deniability. As a countermeasure we target "undeniable deniability" which means that every transcript already contains enough information to be fully deniable. The way we achieve this is through the use of time-lock-puzzles.

Time-lock puzzles (introduced by Rivest, Shamir and Wagner [RSW96]) allow the encryption of a value such that it can be recovered by anyone after performing a certain amount of sequential computation. Before the computation finishes the puzzle only reveals trivial information.

We use the following definition:

Definition 44 (Time Lock Puzzle)**.** A time-lock puzzle TL is a tuple of two PPT-algorithms TL.lock and TL.unlock.

- TL.lock takes three parameters: The security-parameter 1^λ, a duration Δt and a message m and returns a ciphertext c.
- TL.unlock takes the ciphertext c as only parameter and returns a message m.

We require that TL.unlock returns the encapsulated value:

Definition 45 (Correctness for Time Lock Puzzles)**.** A time-lock puzzle is correct if:

$$\Pr\left[\mathsf{TL.unlock}\left(\mathsf{TL.lock}\left(1^\lambda, \Delta t, m\right)\right) = m\right] = 1$$

Furthermore, we require that no adversary can distinguish encapsulated values without performing sequential work for at least Δt. We call this notion INDistinguishability under No-Message-Attacks or IND-NMA for short; or more formally:

Definition 46 (IND-NMA-Security)**.** A timelock-puzzle is IND-NMA-secure, if no PPT-adversaries $\mathcal{A}$ has a chance non-negligibly better than $1/2$ of winning in Experiment 16:

$$\begin{aligned}&\forall \mathcal{A} \in \mathrm{PPT}, \lambda \in \mathbb{N}, \Delta t \in T : \\ &\Pr\left[\mathsf{Exp}^{\mathsf{IND\text{-}NMA}}_{\mathrm{TL},\mathcal{A}}\left(1^\lambda, \Delta t\right) = 1\right] - \frac{1}{2} \\ &=: \mathsf{Adv}^{\mathsf{IND\text{-}NMA}}_{\mathrm{TL},\mathcal{A}}\left(1^\lambda, \Delta t\right) \leq \mathrm{negl}\left(\lambda\right)\end{aligned}$$

Experiment 16: $\mathsf{Exp}^{\mathsf{IND\text{-}NMA}}_{\mathrm{TL},\mathcal{A}}\left(1^\lambda, \Delta t\right)$

$m_0, m_1 := \mathcal{A}(1^\lambda, \Delta t)$
abort if $(|m_0| \neq |m_1|)$
$b \leftarrow_\$ \mathbb{B}$
$c := \mathrm{TL}.\mathsf{lock}\left(1^\lambda,\ \Delta t,\ m_b\right)$
$t_0 := \mathsf{now}()$
$b' := \mathcal{A}(c)$
$t_1 := \mathsf{now}()$
abort if $(t_1 - t_0 \geq \Delta t)$
return $b = b'$

Note also that our definition uses two security parameters: Δt and 1^λ; the latter is a regular security parameter that limits the total amount of (potentially parallel) work an attacker may perform. This is necessary because most time lock puzzles can be broken in a very short amount of time as long as the total work is unbounded (for example by searching for the message and the randomness via parallel brute force).

We note that this definition is simplified compared to pre-existing ones such as the one from Bitansky, Goldwasser, Jain, Paneth, Vaikuntanathan, and Waters [BGJ+15], but sufficient for our purposes: Unlike the more traditional approach that is concerned about the minimum circuit-depth that has to be evaluated before the secret can be learned, ours is considers actual wall-clock

time and thus works on a much higher abstraction-layer. Since the translation from the former to the latter would have to happen at instantiation time anyways, this approach appears to be a better choice in our particular use-case.

The important part here is that parallelism cannot be used to extract the information faster. This means that a consumer PC with a high-end CPU may even be able to overtake super computers. As such the largest thread would be adversaries with very high sequential speed, which could for example be achieved by dedicated implementations in hardware and extreme overclocking. Compared to a simple increase in parallelism, these approaches are much more limited in what they can achieve and how they scale with their cost.

In order to protect our protocol from the attack that we outlined at the start of this subsection, we add time-lock puzzles to it as follows: Instead of just signing $\overline{pk}$ and its expiration-date with $\widehat{sk}$, we encapsulate $\overline{sk}$ in a time-lock puzzle and only sign $\overline{pk}$ and the expiration-date in combination with that puzzle. This way signatures cannot be verified without knowledge that is sufficient to simulate them after expiration, making them unconvincing to every judge. We note that we don't intend for the puzzle to replace the publication of the key, but to supplement it.

We emphasize that we avoid the most common criticism of time-lock-puzzles, namely that they pointlessly waste huge amounts of energy: No honest party in our protocol will attempt to break any puzzle, we only need to ensure that they are *capable* of doing so in principle; the creation of the puzzles on the other hand is usually efficient enough to not be a major concern in that regard.

3.6. Proposal

We propose the scheme depicted in Algorithms 2–5 which, as we will prove, satisfies the security notions that we defined in Section 3.4. In addition to that it actually exceeds our deniability-notion in that a public epoch information $pinfo_e$ can be used to simulate signatures of even the epoch it was released in, as long as enough time has passed.

The core-idea uses the techniques described in the previous section:

- A forward-secure signature scheme $\widehat{\Sigma}$ that is reversed via pebbling as the static layer.

- A regular signature scheme $\overline{\Sigma}$ that is used for the dynamic layer and is replaced with each epoch.
- A pseudorandom value r_e for each epoch that is used to derandomize all non-deterministic algorithms in $\Sigma.\mathsf{evolve}(.)$
- A timelock-puzzle as part of $pinfo_e$ that is used to ensure the reveal of the secret keys after they expire.

We note that all entities that are part of the static scheme will be marked with a "hat"-symbol (such as $\widehat{\Sigma}$), whereas all parts of the dynamic layer will be marked with an overbar (such as $\overline{\Sigma}$).

We use $\mathsf{H}(r, \widehat{pk}||e||n)$ with $n \in \{0, 1, 2\}$ as seeds for the generation of the timelock-puzzles, and the dynamic keys, respectively. We assume that these algorithms take a constant-length seed and stretch it themselves if necessary. We add $\widehat{pk}$ and the epoch e to the message to prevent multi-target attacks.

Our public key pk consists only of the public key $\widehat{pk}$ of the static signature-scheme $\widehat{\Sigma}$. The secret key sk on the other hand is a six-tuple that contains the public key pk, the pebbling-states for both the pseudorandom-seed sk_r and the secret key for the static scheme $\widehat{sk}$, the number e of evolutions that have been performed (initially 0), the current secret key $\overline{sk}$ of the dynamic signature scheme $\overline{\Sigma}$ and the public epoch-information $pinfo_e$ of the current epoch. The last two values are initially set to $\perp$ until $\Sigma.\mathsf{evolve}$ is executed for the first time. For a rough performance-estimate of this scheme, we refer to Section 3.9.

Including the full description of pebbling in our algorithm obscures the more conceptual parts. Hence, we provide two versions: One that uses pebbling as we intend and a simplified (space-inefficient) one that keeps all needed values in a long list. Operations that are only executed in the simplified version are highlighted green and use line-numbers with a postfix "s", whereas operations that only occur in the complete description are highlighted purple and use line-numbers with a postfix „c".

We also note that we hash and evolve $E + V$ times instead of just E times, leading to secrets that are never used; This is just to simplify the definition of $\Sigma.\mathsf{evolve}$ which would otherwise have to treat the first V epochs differently; a real implementation might instead want to add these special-cases. Similarly, most pebbling algorithms would allow to share most of the work between the calls to $\mathsf{PebblePrep}$.

Our $\Sigma.\mathsf{evolve}$-algorithm (Algorithm 3) is completely deterministic since all required randomness is derived from the pseudorandom value r_{new}.

Algorithm 2: Σ.gen. The green operations (line-numbers ending on "s") are only part of the simplified version, the purple ones (line-numbers ending on "c") only of the space-efficient complete one.

1 **fun** $\Sigma.\mathsf{gen}\,(1^\lambda, \Delta t, E, V)$**:**
2 $\quad \widehat{pk}, \widehat{sk}_E := \widehat{\Sigma}.\mathsf{gen}\,(1^\lambda)$
3 $\quad r_E \leftarrow_\$ \mathbb{B}^\lambda$
4s $\quad$ **for** $e \in \{E+V, \dots, 0\}$**:**
5s $\quad\quad \widehat{sk}_e := \widehat{\Sigma}.\mathsf{update}\left(\widehat{sk}_{e+1}\right)$
6s $\quad\quad r_e := \mathsf{H}(r_{e+1}, \widehat{pk}||e+1||0)$
7s $\quad sk_r := [r_{-V}, \dots, r_0, \dots, r_E]$
8s $\quad \widehat{SK} := [\widehat{sk}_{-V}, \dots, \widehat{sk}_0, \dots, \widehat{sk}_E]$
4c $\quad \widehat{SK}_{\text{new}} := \mathsf{PebblePrep}\left(E, \widehat{sk}_E, \widehat{\Sigma}.\mathsf{update}\right)$
5c $\quad \widehat{SK}_{\text{exp}} := \mathsf{PebblePrep}\left(E+V, \widehat{sk}_E, \widehat{\Sigma}.\mathsf{update}\right)$
6c $\quad sk_{r,\text{new}} := \mathsf{PebblePrep}\left(E, (r_E, E), (r, e) \to (\mathsf{H}(r_{e+1}, \widehat{pk}||e+1||0), e-1)\right)$
7c $\quad sk_{r,\text{exp}} := \mathsf{PebblePrep}\left(E+V, (r_E, E),\right.$
$\quad\quad \left.(r, e) \to (\mathsf{H}(r_{e+1}, \widehat{pk}||e+1||0), e-1)\right)$
9 $\quad t_0 := \mathsf{now}()$
10 $\quad pk := (\widehat{pk}, t_0, \Delta t, E, V)$
11s $\quad sk := (pk, sk_r, \widehat{sk}, 0, \bot, \bot)$
11c $\quad sk := (pk, (sk_{r,\text{new}}, sk_{r,\text{exp}}), (\widehat{SK}_{\text{new}}, \widehat{SK}_{\text{exp}}), 0, \bot, \bot)$
12 $\quad$ **return** pk, sk

Signing messages (Algorithm 4) works by signing the message and the public epoch information $pinfo_e$ with the $\widehat{sk}$. We sign $pinfo_e$ to ensure that verification only works for parties who know it, which aims at increasing the deniability.

Our verification algorithm (Algorithm 5) checks whether the signature is not yet expired and whether it was valid in the epoch in which it was generated. If and only if both of these requirements are fulfilled, the signature is accepted. The reason for why we prepend $\widehat{pk}||t$ to the arguments of all calls to H is simply to prevent multi-target attacks. While our proof does not make use of this and as such there is no effect on our final security statement, we consider it good practice to do so anyway.

Theorem 4. *Σ is complete in the sense of Definition 41.*

Proof. Given the structure of the completeness-game and the conditions of its abort-case, the challenge-signature is less than V epochs old and the current epoch e is not past the lifetime of the key ($e < E$). Furthermore, the structure of the game also implies that the epoch of the challenge-signature is not a future one.

Given that the game can only be won if either b_0 or b_1 is set to false.

b_0 is the result of the verification of $\hat{\sigma}$ under $\widehat{pk}$. $\hat{\sigma}$ was honestly generated for the exact message that is now being verified. The completeness of $\widehat{\Sigma}$ therefore implies that $b_0 = 1$.

b_1 is the result of the verification of $\overline{\sigma}$ under $\overline{pk}$. $\overline{\sigma}$ was honestly generated for the exact message that is now being verified. The completeness of $\overline{\Sigma}$ therefore implies that $b_1 = 1$.

Combined this means that $\neg(b_0 \wedge b_1)$ is always false, therefore there is no adversary with a higher chance than 0 of winning the completeness-game, therefore Σ is complete. □

Theorem 5. *Σ is unforgeable in the sense of Definition 42 with:*

$$\begin{aligned}&\mathsf{Adv}^{\mathit{EEUF\text{-}CMA}}_{\Sigma,E,V,\mathcal{F}}(1^\lambda,\ \Delta t)\\ \leq E\cdot&\left(\begin{array}{l} E\cdot \mathsf{Adv}^{\mathit{PRF}}_{\mathsf{H},\mathcal{A}_{\mathit{PRF}}}(1^\lambda)\\ +\ V\cdot \mathsf{Adv}^{\mathit{IND\text{-}NMA}}_{TL,\mathcal{A}_{\mathit{IND\text{-}NMA}}}(1^\lambda,\ V\cdot\Delta t)\\ +\ \mathsf{Adv}^{\mathit{FS\text{-}EUF\text{-}CMA}}_{\widehat{\Sigma},E,\mathcal{A}_4}(1^\lambda)\\ +\ \mathsf{Adv}^{\mathit{EUF\text{-}CMA}}_{\overline{\Sigma},\mathcal{A}_5}(1^\lambda)\end{array}\right)\end{aligned}$$

Proof. Intuitively we use game-hopping, with the regular EEUF-CMA-game as starting-point.

In the first hop we guess the epoch e for which $\mathcal{F}$ will present a forgery and abort the execution after $e + V - 1$ epochs (loss-factor of $\frac{1}{E}$).

In the next hop we replace the random seeds r_e as well as the pseudorandom random-tapes used for TL.lock and $\overline{\Sigma}$.gen of all epochs including and after e with random values, which works because of the PRF-security of H (loss $\leq E\cdot \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{H},\mathcal{A}}(1^\lambda)$).

In the next hop we encapsulate random values in all timelock-puzzles that are generated in and after epoch e. This works because of their hiding-property and because the game enforces that $\mathcal{F}$ doesn't have enough time to unlock them (loss $\leq V\cdot \mathsf{Adv}^{\mathsf{IND\text{-}NMA}}_{TL,\mathcal{A}}(1^\lambda,\ V\cdot\Delta t)$).

In the next hop we check whether the forged signature contains a fresh signature under $\widehat{pk}$, present it to an FS-EUF-CMA-challenger and abort the game if it is (loss $\leq \mathsf{Adv}^{\mathsf{FS\text{-}EUF\text{-}CMA}}_{\widehat{\Sigma},E,\mathcal{A}}(1^\lambda)$).

In the last hop we note that the forged signature must contain a fresh signature under $\overline{pk}$ and we present it to an EUF-CMA-challenger (loss $\leq \mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\overline{\Sigma},\mathcal{A}}(1^\lambda)$).

In Detail:

Let E be the total number of epochs. We will show that our scheme is secure using game-hopping:

Let 0 be the original EEUF-CMA-game and $\Pr[\text{break}_0]$ be the probability that a forger $\mathcal{F}$ succeeds in presenting a forgery.

For 1 we guess the epoch e for which the forger will be successful in creating a forgery and abort if this is not the case. Since signatures are valid for V epochs, this means that this abort will happen by the end of epoch $e + V$. Our guess will be correct with probability E^{-1}, giving us:

$$\Pr[\text{break}_0] \leq E \cdot \Pr[\text{break}_1]$$

In 2 we replace the results of all evaluations of $\mathsf{H}(r_e, \cdot)$ with random values. In order to do this, we have to replace all later values of r as well in reverse order, using the PRF-assumption for H.

Let *Game 2.E* := *Game 1*. For all i in $\{E-1, \ldots, e\}$ perform the following replacement: (Note that the second number in the game labels identifies the epoch that is modified and is thus counting down instead of up.)

In *Game 2.i* we replace the results of all evaluations of $\mathsf{H}(r_{i+1}, \cdot)$ with random values. To show that this replacement is sound we initialize a PRF-challenger and use the oracle it provides instead of computing H directly. This replacement is sound as by *Game 2.i*+1 r_{i+1} is a truly random value. If the challenger's internal bit is 0 then we are in *Game 2.*$(i+1)$, otherwise we are in *Game 2.i*. We can therefore convert any adversary capable of detecting this change into an adversary $\mathcal{A}_{2.i}$ capable of breaking the PRF-security of H:

$$\Pr[\text{break}_{2.i+1}] \leq \Pr[\text{break}_{2.i}] + \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{H},\mathcal{A}_{2.i}}(1^\lambda)$$

Since there are at most E hops like this, we conclude that:

$$\Pr[\text{break}_1] \leq \Pr[\text{break}_2] + E \cdot \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{H},\mathcal{A}_2}(1^\lambda)$$

In 3 we encapsulate random strings in $tl_e, \ldots, tl_{e+V-1}$ instead of the secrets in question. To do this we need V almost identical sub-hops. Let *Game 3.V* := *Game 2*. For all i in $\{V-1, \ldots, 0\}$ perform the following replacement: (Note that the second number in the game labels is again counting down.)

In *Game 3.i* we replace the value encapsulated in tl_{e+i} with a random string. This is a sound replacement due to the temporal hiding property of the time lock puzzle: To show this we initialize a hiding-challenger and query it with $r_t||\widehat{sk}_t$ as m_0 and a random string of the same length as m_1. This replacement is sound since the randomness used to generate the time lock puzzle is actually random by *Game* $3.(i+1)$ and since the overall security-game ensures that the puzzle is not older than $V \cdot \Delta t$. If the challenger's internal bit is 0 then we are in *Game* $3.(i+1)$. Otherwise, the output is the encapsulation of a random value, and we are in *Game* $3.i$. Any adversary capable of distinguishing these two games can therefore be turned into an adversary $\mathcal{A}_{3.i}$ against the hiding property of the time lock puzzle and we have:

$$\Pr\left[\text{break}_{3.(i+1)}\right] \leq \Pr\left[\text{break}_{3.i}\right] + \mathsf{Adv}^{\mathsf{IND\text{-}NMA}}_{TL,\mathcal{A}_{3.i}}\left(1^\lambda,\ V \cdot \Delta t\right)$$

By defining *Game 3* := *Game 3.0* and combining these sub-hops, we get:

$$\Pr\left[\text{break}_2\right] \leq \Pr\left[\text{break}_3\right] + V \cdot \mathsf{Adv}^{\mathsf{IND\text{-}NMA}}_{TL,\mathcal{A}_3}\left(1^\lambda,\ V \cdot \Delta t\right)$$

In 4 we modify the key-generation as follows: After setting up r and sk_r we compute $r_{e'}$, $\overline{pk}_{e'}$, $\overline{sk}_{e'}$, $tl_{e'}$ for all epochs $e' \in \{e - V + 1, \ldots, e + V\}$ directly. After that we initialize a FS-EUF-CMA-challenger for the forward-secure signature scheme and receive its public key $\widehat{pk}$ which we will use instead of generating our own. Then we request the signatures on our dynamic level public keys. For this we first ask the challenger to perform $E - e - V + 1$ key-updates. For each $i \in \{V - 1, \ldots, 0\}$ we then

- request a signature $\hat{\sigma}_{e+i}$ on $\overline{pk}_{e+i}\ ||e+i\ ||r_{e+i-V}\ ||tl_{e+i-V}$ and then
- request a key-update.

Then we request the updated secret key $\widehat{sk}_{e-1}$. Using this key we can set up $\widehat{sk}$ for the first $e-1$ epochs and complete the key-generation as normal. For the first $e-1$ epochs we leave $\Sigma.\mathsf{evolve}$ and $\Sigma.\mathsf{sign}$ unchanged. From the e'th epoch onwards we modify $\Sigma.\mathsf{evolve}$ to use the respective $\hat{\sigma}_{e'}$ and $tl_{e'}$ that we prepared during the key-generation instead of computing them honestly and set $\widehat{sk}$' to $\perp$ as it is no longer needed: By *Game 1* no more evolutions are performed after the $e + V$'th epoch and $\Sigma.\mathsf{sign}$ does not use that key. All values that are given to the adversary are still sampled from the same distributions as in *Game 4*, therefore this part of the change is perfectly undetectable. If the adversary presents a valid forgery for epoch

e, we check whether the static level signature is equal to $\hat{\sigma}_e$. If it is not, we can forward it to the FS-EUF-CMA-challenger and win the FS-EUF-CMA-game and abort. Otherwise, we proceed as before. The difference between the games is therefore perfectly indistinguishable unless $\mathcal{F}$ manages to forge a static level signature. In other words any adversary capable of detecting this change can be converted into an adversary $\mathcal{A}_5$ capable of breaking the FS-EUF-CMA-security of $\widehat{\Sigma}$:

$$\Pr[\text{break}_3] \leq \Pr[\text{break}_4] + \mathsf{Adv}^{\mathsf{FS\text{-}EUF\text{-}CMA}}_{\widehat{\Sigma}, E, \mathcal{A}_4}(1^\lambda)$$

In 5 we lastly consider the case in which the adversary attacks the dynamic scheme: When it is time to compute $\overline{pk}_e$ and $\overline{sk}_e$ we instead initialize an EUF-CMA challenger for the dynamic signature-scheme and use the challenge public key as $\overline{pk}_e$. This replacement is sound as the randomness used for the key-generation algorithm is true randomness by *Game 2*. Whenever we need to sign a message using $\overline{sk}_e$ we instead request the signature from the challenger. Once the forger presents us with a valid forgery the static level signature is by *Game 4* the one that we created during the key-generation phase and therefore not new. Since the new signature is by assumption fresh this means by the structure of the scheme, that the dynamic level signature is fresh. Therefore, it can be forwarded to the EUF-CMA challenger who will accept it. Therefore, any adversary capable of winning *Game 5* can be converted into an adversary $\mathcal{A}_6$ that has the same probability of breaking the dynamic signature scheme and we find:

$$\Pr[\text{break}_5] = \mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\overline{\Sigma}, \mathcal{A}_5}(1^\lambda)$$

Combining these, we get that for all forgers $\mathcal{F}$:

$$\begin{aligned}
&\mathsf{Adv}^{\mathsf{EEUF\text{-}CMA}}_{\Sigma, E, V, \mathcal{F}}(1^\lambda, \Delta t) \\
\leq & E \cdot \left(\begin{array}{l} E \cdot \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{H}, \mathcal{A}_{\mathsf{PRF}}}(1^\lambda) \\ + \ V \cdot \mathsf{Adv}^{\mathsf{IND\text{-}NMA}}_{TL, \mathcal{A}_{\mathsf{IND\text{-}NMA}}}(1^\lambda, \ V \cdot \Delta t) \\ + \ \mathsf{Adv}^{\mathsf{FS\text{-}EUF\text{-}CMA}}_{\widehat{\Sigma}, E, \mathcal{A}_4}(1^\lambda) \\ + \ \mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\overline{\Sigma}, \mathcal{A}_5}(1^\lambda) \end{array} \right)
\end{aligned}$$

Where $\mathcal{A}_{\mathsf{PRF}}$ is the adversary with the highest advantage among all $\mathcal{A}_{2.i}$ and $\mathcal{A}_{\mathsf{IND\text{-}NMA}}$ is the adversary with the highest advantage among all $\mathcal{A}_{3.i}$. This means that if H, TL, $\widehat{\Sigma}$ and $\overline{\Sigma}$ provide the respective security notions, then Σ is unforgeable in the sense of EEUF-CMA. □

Theorem 6. Σ *is deniable in the sense of Definition 43.*

Proof. Our simulator $\mathcal{S}$ uses the information in $pinfo_e$ to create a secret key that is equivalent to the real one for all expired epochs and then simply executes the signing algorithm as an honest party would. We first introduce Algorithm 6 which extracts a suitable secret key from the public epoch information $pinfo_e$.

With this the actual simulation (Algorithm 7) essentially just executes $\Sigma.\mathsf{sign}$. If the sk that is computed by the simulator is indeed equivalent to the real secret key, the deniability of our scheme follows immediately from the remaining definition of the simulator and the fact that $\Sigma.\mathsf{evolve}$ is deterministic. To see that sk is in fact equivalent it is enough to see that the only difference between it and the real key is that the pebbling data structure doesn't go back as many key/randomness evolutions, preventing the use in future epochs. By the structure of the game, this information is however not needed at the point at which $\mathcal{S}$ runs. Therefore, the only difference is one that does not make a difference for the generated signatures because sign does not use that information. Therefore, the keys are equivalent for all past epochs and the simulated signatures are distributed exactly as they would be if they were honestly generated. Since $pinfo_e$ and the relevant parts of sk_e are identical to the ones an honest party would have used and since $\overline{\Sigma}$ is assumed to be stateless, this means that the resulting signature is clearly information-theoretically indistinguishable from a real one. Because this perfect indistinguishability holds even if $\mathcal{S}$ is called more than once there is no way for $\mathcal{J}$ to learn b, limiting her to guessing a random bit, which has a success-probability of $\frac{1}{2}$.

Therefore Σ is perfectly offline deniable. □

Corollary 6.1. *If* $\overline{\Sigma}$ *is a deterministic signature-scheme, the simulated signatures are identical and not just indistinguishable from real ones.*

We would like to add the following strengthening to Theorem 6: Even if the simulator only receives $pinfo_e$ when it should also create a signature for epoch e, it is still possible to create a perfectly indistinguishable signature. To do so, $\mathcal{S}$ starts by opening the time-lock puzzle tl_e (part of $pinfo_e$) and will after performing computations for roughly $V \cdot \Delta t$ time receive r_e and $\widehat{sk}_e$. With those he can execute Algorithm 7 as before and the resulting signatures will be perfectly indistinguishable for the same reason presented above as well.

3.7. Epochal Group Chat

Having demonstrated the viability of epochal signatures, we can now demonstrate their usefulness in the context of deniable group-chats:

Theorem 7. *Let* Π *be a chat-protocol for which the following requirements hold:*

1. *A hidden full interaction (*HFI*) causes a perfect state disassociation.*
2. Π *only uses the secret key of a simple (EUF-CMA-secure) signature-scheme as long-term secret.*
3. Π *works with every EUF-CMA-secure signature scheme.*
4. Π *only uses the long-term secret key to create signatures with the regular signing algorithm.*
5. *There exists a time period* t_Π *such that* Π *never verifies a signature more than* t_Π *after its creation.*
6. Π *does not use any oracles that cannot be efficiently simulated.*

Then the protocol Π^* *that only differs from* Π *in that the conventional signature-scheme is replaced with an epochal signature-scheme* Σ *with parameters so that* $(V-1)\cdot\Delta t \geq t_\Pi$ *is HFI-OfD-secure and offers the same completeness and authenticity as* Π*.*

Proof. This follows directly from Theorems 8, 9, and 10. □

The theorem essentially states that if the protocol uses generic EUF-CMA signatures and is HFI-OfD-secure when these signatures are removed then the protocol obtained by replacing the signatures with epochal signatures is HFI-OfD-secure.

We need Requirements 1 and 2 to ensure that nothing besides the signatures prevents Π from being deniable.

The purpose of Requirement 3 is to ensure that the signature is not used in a way that requires more structure than is guaranteed by EUF-CMA. A protocol that assumes that the signature has the form of a group element, that is then used as such in further computations can for example not be generically replaced with any epochal signature scheme and is therefore excluded by this requirement.

This is similar to Requirement 4, which exists to ensure that no entropy of the secret key, that may be inaccessible to the simulator is used to modify behaviour that differs from the simulator who does not know that key.

Requirement 5 is necessary to ensure that the protocol keeps working with epochal signatures, as their expiration would otherwise mean that signatures that should be successfully verified would fail in that verification.

Requirement 6 is necessary to allow the undetected modification of the game during the proof: In some instances it may be necessary to call those oracles at different times than in a real execution which might allow the judge to detect changes. We don't believe that this will cause problems for most protocols, and expect that it might be possible to weaken this requirement, but it is a necessary restriction.

Many contemporary protocols use authentication-means different from signatures, for example to ensure that the protocols in question are deniable. Because of this we are not aware of a currently widely used protocol that meets our definition, but note that MLS qualifies and is indeed the primary target for this substitution.

3.7.1. Completeness

Theorem 8. Π^* *offers the same completeness as* Π.

Proof. Since Requirement 5 ensures that all signatures are verified before they are expired and since EEUF-CMA is essentially identical to EUF-CMA before that and since Requirement 3 ensures that replacing the signature scheme will not cause breakage in other parts of the protocol, Π^* retains the completeness of Π. □

3.7.2. Authenticity

Theorem 9. Π^* *offers the same authenticity as* Π.

Proof. The authenticity of Π^* follows from the authenticity of Π and Requirement 5: Since all epochal signatures are verified within their validity-period during which their unforgeability-notion is essentially identical to EUF-CMA every security-argument for authenticity works completely analogous to the way it works for Π. □

3.7.3. Deniability

Intuitively the deniability of the scheme is primarily a consequence of the deniability of the epochal signature-scheme, Requirement 2 that all other secrets are ephemeral and can thus be sampled by a simulator and Requirement 1 that demands that Π is disjoined under hidden full interactions. Most of the other requirements boil down to "the protocol does nothing weird or unusual" and are therefore less important for a first intuition.

In order to define our simulator $\mathcal{S}$ we first introduce the utility-algorithm update_group (Algorithm 8) that $\mathcal{S}$ will use to separate different ch-segments and disassociate the state. Intuitively it has every party that stays in the group send an empty message (ε) and adds and removes all other users as needed.

We note that update_group does not use the secret keys that are provided as part of the state but instead receives them as an explicit argument. This is in order to allow the use of keys that are, for the purposes of the simulation, equivalent to the real ones but not necessarily identical.

We add as Lemma 3 that update_group causes a state disassociation in the target-group:

Lemma 3. *The algorithm* update_group*, as defined in Algorithm 8, when called with a starting state st, an update-group-description gr', a list of secret-keys SK and a timepoint time, causes a state disassociation in the group identified by $ID_{gr'}$.*

Proof. It follows from the definitions that update_group causes a hidden full interaction in that group. With that the statement follows directly from Requirement 1. □

This leads to our main-theorem of this section:

Theorem 10. Π^* *offers HFI-OfD-deniability under the following conditions, where e is the last epoch for which the challenge instruction list contains an action:*

- *The simulator has access to $pinfo_e$ for all parties.*
- *The last* ch*-action that $\mathcal{S}$ has to simulate occurs in epoch $e - V$ or earlier.*

Proof. We define $\mathcal{S}$ as depicted in Algorithm 9, where we summarize the $pinfo_e$'s of all parties in the set PI_e that contains them in tuples with the matching public-keys.

The only information that $\mathcal{S}$ receives beyond what he is given in the deniability game is PI_e, which is in line with Condition 10. We consider this condition plausible, as we require that all public epoch information is made publicly available. Formally this could be modelled as putting it onto a global bulletin-board with unrestricted read-access.

As exec is the only black-box algorithm that $\mathcal{S}$ executes and since we exec is efficiently computable (formally: $\mathsf{exec} \in \mathrm{PPT}$) $\mathcal{S}$ clearly runs in polynomial time with regards to $\mathsf{max}\left(\lambda, |sim_{il,st}|\right)$ and is thus itself a PPT-algorithm.

With this we now show that the transcript generated by $\mathcal{S}$ is information-theoretically indistinguishable from a real one. For this we will use game-hopping: We start by looking at the output-distribution of $\mathcal{J}$ at the end of the game, starting with the case where the entire transcript that she receives is real ($b = 0$). After this we will modify the game that $\mathcal{J}$ plays in a way that is undetectable by her, until the transcript she receives is partially generated by $\mathcal{S}$ ($b = 1$). Since $\mathcal{J}$ is unaware of any of these changes, her output-distribution is unchanged, which means that it is independent of b. This in turn implies that her chance of answering with the correct value of b cannot be higher than achievable by random guessing and that Π^* is therefore $\mathsf{HFI\text{-}OfD}$-secure. We remark here that looking at $\mathcal{J}$'s outputs instead of the return-value of the experiment may appear unusual but is intentional.

Let ***Game 0*** be the $\mathsf{HFI\text{-}OfD}$-game with $b = 0$, aka the game in which the entire transcript is real. Let furthermore $\mathrm{OutDist}_{\mathcal{J},0}$ be the distribution of the judge's guess for b.

Let e be the epoch in which il contains the last `ch`-action. In ***Game 1*** we delay the execution of all actions in the $\mathsf{HFI\text{-}OfD}$-game until the e'th epoch, then perform them all without delays. This can be done as Requirement 6 allows the challenger to simulate any oracles that the protocol might use efficiently. As the actual output of the protocol is not affected by this change, we get $\mathrm{OutDist}_{\mathcal{J},1} = \mathrm{OutDist}_{\mathcal{J},0}$.

In ***Game 2*** we execute $il_{\texttt{ch}}$ twice in parallel. We will henceforth call these the "left" and the "right" execution. Initially only the left execution is used in the actual game, that is both the `exp`- and the `ch`-transcript that $\mathcal{J}$ gets are taken from its execution. Over the next game hops we will modify these executions so that the left one becomes an honest execution of $il_{\texttt{ar}}$ and that the right one becomes the simulator. So far this is however a perfectly undetectable change from $\mathcal{J}$'s perspective and we get $\mathrm{OutDist}_{\mathcal{J},2} = \mathrm{OutDist}_{\mathcal{J},1} = \mathrm{OutDist}_{\mathcal{J},0}$.

In ***Game 3*** we modify the game so that the transcript of the `ch` actions is taken from the right execution of il, whereas the transcript of the `exp` actions is still taken from the left. We perform this change on a per-group basis, with

one sub-game per group. Let *Game 3.0* := *Game 2*, then for each group gr_i we define *Game 3.i* as follows:

Let n be the number of longest segments in *il*, in which group gr_i contains only `ch` actions. We will now call each of these segments a `ch`-segment. With this we now perform n sub-sub-games to replace the left transcripts of these `ch`-segments with the right transcripts. Let *Game 3.i.n* = *Game 3.*$(i-1)$*.0* and *Game 3.1.n* = *Game 3.0*.

In *Game 3.i.j* we replace the left transcript of the j'th `ch`-segment with the right transcript. We point out that by the definition of the HFI-OfD-game every `ch`-segment is embraced by `hid`-segments that contain hidden full interactions. Intuitively we will flip the left and the right thread of execution twice, so that the right one is on the left side during the `ch` segment and the points of contact are in the `hid` segments.

We will use two essentially identical sub-sub-sub-hops to show that this change is undetectable: First we use the left thread of execution as the right one (and the other way around) after the completion of the i'th `ch` segment (this changes their roles from then on). To show that this replacement is sound we initialize a state disassociation challenger $\mathcal{C}$ and give it the states that result on the left and the right from executing the `ch` segment as well as the instruction sub list of following `hid` segment, gr_i and *SK* and proceed with the execution using $\mathcal{C}$'s first output for the left side and the second output for the right side.

This replacement is sound because both states are by the definition of the game equivalent and the instruction-lists are identical and consistent. If $\mathcal{C}$ samples $b = 0$ then this we are in *Game 3.i.*$(j-1)$, otherwise in *Game 3.i.j.1*, getting $\mathrm{OutDist}_{\mathcal{J},3.i.j.1} = \mathrm{OutDist}_{\mathcal{J},3.i.(j-1)} = \mathrm{OutDist}_{\mathcal{J},0}$.

We repeat the exact same replacement for the `hid` segment before the `ch` segment and get $\mathrm{OutDist}_{\mathcal{J},3.i.j} = \mathrm{OutDist}_{\mathcal{J},3.i.j.1} = \mathrm{OutDist}_{\mathcal{J},0}$ We remark that the first and the last `ch` segment might not be preceded/succeeded by a `hid` segment; in that case we simply leave out the respective sub-sub-sub-hop, which does exactly what we want in that case.

Let $|st.\mathbf{G}| =: m$, then we define *Game 3* := *Game 3.m.0* and find that: $\mathrm{OutDist}_{\mathcal{J},3} = \mathrm{OutDist}_{\mathcal{J},2} = \mathrm{OutDist}_{\mathcal{J},0}$.

In ***Game 4*** we reconnect the `ch`-segments in the right execution by using `update_group` (Algorithm 8) instead of the non-`ch`-actions. We do this on a per-group basis, with one sub-game per group. Let *Game 4.0* := *Game 3*, then for each group gr_i we define *Game 4.i* as follows:

Let n be the number of `ch`-segments in group gr_i. We perform n sub-sub-games to connect these directly with each other. Let *Game 4.i.n* =

Game $4.(i-1).0$ and *Game* $4.1.n$ = *Game* 4.0 and note that we again count the last variable downwards.

Let st be the state that results from executing the last instruction of the $j-1$'th `ch`-segment of gr or the empty state if no such state exists. Let st' be the state before the first instruction of the j'th `ch`-segment. In *Game* $4.i.j$ we replace st' with st'' which we define to be the result of executing $\mathsf{update\_group}(st, st'^{gr_i}.ps, SK)$

To show that this replacement is sound, we initialize a state disassociation challenger $\mathcal{C}$ and call it with st, il_0, st, il_1 and gr_i, where il_0 and il_1 are defined as follows: The instruction list il_0 contains all non-`ar` instructions between st and st'. The instruction list il_1 is the output of `update_group` when called with the arguments stated above. We then replace st' with the first state that $\mathcal{C}$ outputs. This replacement is sound as il_0 contains a hidden full interaction by the definition of the HFI-OfD game, il_1 contains a hidden full interaction by the definition of `update_group`, both are consistent with their starting states and result by construction in equivalent states. If $\mathcal{C}$'s bit b is 0, then the resulting first state was generated like the one in *Game* $4.i.(j-1)$ and we are in that game. If $\mathcal{C}$'s bit b is 1, then the resulting first state was generated like st'' and we are in *Game* $4.i.j$. Since Π^* is by Requirement 1 perfectly separated under hidden full interactions, we get: $\mathrm{OutDist}_{\mathcal{J},4.i.j} = \mathrm{OutDist}_{\mathcal{J},4.i.j-1} = \mathrm{OutDist}_{\mathcal{J},0}$.

Let $|st.\mathbf{G}| =: m$, then we define *Game* $4 :=$ *Game* $4.m.0$ and find that: $\mathrm{OutDist}_{\mathcal{J},4} = \mathrm{OutDist}_{\mathcal{J},3} = \mathrm{OutDist}_{\mathcal{J},0}$.

In ***Game 5*** we modify the left execution to perform the `ar`-actions instead of the `ch`-actions. We do this on a per-group basis, with one sub-game per group. Let *Game* $5.0 :=$ *Game* 4, then for each group gr_i we define *Game* $5.i$ as follows:

Let n be the number of `exp`-segments in group gr_i. We perform n sub-sub-games to connect these directly with each other. Let *Game* $5.i.n =$ *Game* $5.(i-1).0$ and *Game* $5.1.n =$ *Game* 5.0 and note that we again count the last variable downwards.

Let st be the state that results from executing the last instruction of the $j-1$'th `exp`-segment of gr or the empty state if no such state exists. Let st' be the state before the first instruction of the j'th `exp`-segment. Let il^* be the instruction list between st and st' (this means that at this moment the game executes $il^*_{\mathtt{ch}}$). In *Game* $5.i.j$ we replace the st' with st'' which we define to be the result of executing $il^*_{\mathtt{ar}}$ instead of $il^*_{\mathtt{ch}}$.

To show that this replacement is sound, we initialize a state disassociation challenger $\mathcal{C}$ and call it with st, $il^*_{\mathtt{ch}}$, st, $il^{*\prime}_{\mathtt{ar}}$ and gr_i, where $il^{*\prime}_{\mathtt{ar}}$ is identical to $il^*_{\mathtt{ar}}$, except that the types of the `ch`- and `ar`-instructions are switched. We

then replace st' with the state that $\mathcal{C}$ outputs. This replacement is sound as both executable instruction lists contain a hidden full interaction by the definition of the HFI-OfD game, both are consistent with their starting states and result by construction in equivalent states. If $\mathcal{C}$'s bit b is 0, then the resulting state was generated like the one in *Game 5.i.*$(j-1)$ and we are in that game. If $\mathcal{C}$'s bit b is 1, then the resulting state was generated like st'' and we are in *Game 5.i.j*. Since Π^* is by Requirement 1 perfectly separated under hidden full interactions, we get: $\text{OutDist}_{\mathcal{J},5.i.j} = \text{OutDist}_{\mathcal{J},5.i.j-1} = \text{OutDist}_{\mathcal{J},0}$.

Let $|st.\mathbf{G}| =: m$, then we define *Game 5* := *Game 5.m.0* and find that: $\text{OutDist}_{\mathcal{J},5} = \text{OutDist}_{\mathcal{J},4} = \text{OutDist}_{\mathcal{J},0}$.

In ***Game 6*** we extract secret keys from the final public epoch information PI_e using extract_sk (Algorithm 6). The resulting secret keys are equivalent for the first $e - V$ epochs, which are by assumption the only epochs in which secret keys are used as a result of ch-statements.

To show that this replacement is sound, we use multiple sub-games: Let *Game 6.0* be equal to *Game 5*.

In *Game 6.i* we replace the i'th signature σ that the challenger has to generate with for the ch-transcript with a simulated one. To do so we extract a secret key from the public epoch information $pinfo_e$ of the signer and use extract_sk (Algorithm 6 to acquire a secret key that is equivalent to the real one for the first $e - V$ epochs.) To show that this replacement is sound, we initialize a deniability-challenger $\mathcal{C}$ (running at regular time) against Σ and request a key-pair that we will use for the party $\mathcal{P}$ issuing that signature instead of generating a fresh it ourselves. Let m be the message that σ has to sign and e' the epoch in which it has to be generated. We give the tuple (m, e', V) to $\mathcal{C}$ who will output a challenge-signature just in time for its use in the protocol execution (as that execution is delayed by V epochs). If $\mathcal{C}$'s challenge-bit is 0, then the signature is real and we are in *Game 6.i* $- 1$, otherwise it is simulated and we are in *Game 6.i*. Since Σ offers perfect deniability, distinguishing these cases is impossible and we find $\text{OutDist}_{\mathcal{J},6.i} = \text{OutDist}_{\mathcal{J},6.(i-1)} = \text{OutDist}_{\mathcal{J},0}$.

By defining *Game 6* := *Game 6.n* we therefore find that $\text{OutDist}_{\mathcal{J},6} = \text{OutDist}_{\mathcal{J},5} = \text{OutDist}_{\mathcal{J},0}$.

In ***Game 7*** we note that the simulation at this point only uses the long-term public-keys PK, the public epoch-informations of the last epoch PI_e, the ch-instructions and the partial group states that precede them if they are preceded by a non-ch-instruction in the group in question. The later of these are exactly equivalent to the simulation instructions based on the starting state and the overall instruction list. By separating this part from the rest

of the HFI-OfD-challenger, we get exactly the simulation-algorithm $\mathcal{S}$ that we defined in Algorithm 9. This is just a conceptual change, so $\text{OutDist}_{\mathcal{J},7} = \text{OutDist}_{\mathcal{J},6} = \text{OutDist}_{\mathcal{J},0}$.

Finally, in ***Game 8*** we note that the final state that is given to $\mathcal{J}$ is generated by running $il_{\mathtt{ar}}$. This means that this game is in fact identical to the original HFI-OfD-game when $b = 1$. Since this is again just a conceptual change we get $\text{OutDist}_{\mathcal{J},8} = \text{OutDist}_{\mathcal{J},7} = \text{OutDist}_{\mathcal{J},0}$.

From this we conclude that the distribution of $\mathcal{J}$'s outputs in the HFI-OfD-game is independent of the value of b and that Π^* is therefore HFI-OfD-secure. □

3.8. Considerations

3.8.1. Offline Users

Given that our protocol offers deniability by releasing secrets that can be used to forge transcripts of past communication, achieving message authenticity with offline-users is inherently difficult. Given that part of the motivation for deniability is the desire to have online-chats behave similarly to chats in the real-world, we considered what the equivalent for offline users might be and did indeed find a sensible counterpart: Imagine that there is supposed to be a group-meeting between Alice, Bob, Carol and Dave; Alice is however held up elsewhere and cannot attend in person.

What would happen in practice is that one of the attending people, say Bob, would tell the actual contents to Alice after the meeting is over. While Bob could in principle lie to Alice, she can confirm Bobs story with Dave and Carol. If all of them agree, their story is not only very likely, but even *equivalent* to the truth: In principle the three could always meet before the meeting, discuss what they want to happen during the meeting and then act that out during the actual meeting; since Alice is by assumption not in the meeting, she has also no way of detecting their farce by behaving in an unexpected way.

In our digital setting, this already gives us a manual of how to proceed: When Alice comes back online, Bob will send her the transcript of the chat since Alice went offline and Dave and Carol will confirm it by sending her a hash of it. Alice computes the hash of the transcript as well and if they all match, she knows that Bob, Dave and Carol all agree that this is the story that Alice is supposed to learn.

3.8.2. Expiring Authorisation

Our epochal signature scheme has the ability to create secret keys that can be used for a predetermined number of epochs in a way that is entirely transparent for the verifier. (This is done by simply removing all information about the later epochs from the secret key.) For a possible use case consider a user $\mathcal{U}$ who plans to enter another country and is worried that her device might be seized. Using an entirely fresh key for this could leak to her communication partners that she is traveling. Using a reduced key on the other hand would prevent that information leakage while still limiting impersonation-attacks in case her device is seized to the epochs in which the reduced key is valid. We remark that this is clearly just a measure to reduce damage instead of preventing it and that the use of a fresh key is preferable if the aforementioned leakage is acceptable. However, in case it is not, the use of a reduced key is better than using the full key. While not fully preventing impersonation, it allows to warn communication partners that the key has been corrupted but will become "uncorrupted" again after the last compromised epoch.

3.9. Performance Estimates

The runtime of $\Sigma.\mathsf{gen}$ essentially consists of the time to run $\widehat{\Sigma}.\mathsf{gen}$ and preparing the pebbling-structures. $\Sigma.\mathsf{evolve}$ consists of four pebbling operations (two for key-evolution and two for evolving the pseudorandom r values), one generation of a timelock-puzzle, two calls to H, and one call to $\overline{\Sigma}.\mathsf{gen}$ and $\widehat{\Sigma}.\mathsf{sign}$, each. The computation of the later two is not necessary for fast-forwarding. $\Sigma.\mathsf{sign}$ roughly consists of one call to $\overline{\Sigma}.\mathsf{sign}$, and $\Sigma.\mathsf{verify}$ of one call to $\widehat{\Sigma}.\mathsf{verify}$ and $\overline{\Sigma}.\mathsf{verify}$, each.

To give a performance estimate we instantiate these primitives as follows: We use 2^{20} epochs, consisting of 5 minutes each, resulting in a key-validity of 9.97 years. We use a forward-secure implementation [HKS] of XMSS as described in RFC 8391, specifically the XMSS-SHA2_20_256 variant, as static signature scheme. We use the classical RSA-based timelock-puzzle [RSW96] with 2048 bit modulus and (to simplify measurements) RSA-2048 bit signatures as dynamic signature scheme. We assume the use of an optimal speed-1 pebbling algorithm as well as the use of SHA256 as H. All of the following measurements were performed on a Ryzen 3600 and (except for the XMSS benchmarks) using openssl 1.1.1h.

For XMSS-SHA2_20_256 the average signing time measured is 4.23 ms, which also updates the key and thus doubles as key-evolution-time. Signa-

ture verification takes 0.25 ms on average, and key generation takes about 7.7 min, which does however generate all values that are necessary to set up the pebbling structure without additional cost. (Key generation time of XMSS can be brought down to 0.9 s when using the multi-tree version. This comes at the cost of doubled signature size and - more importantly - using a non-generic multiple layer structure for the pebbling.)

For RSA-2048 we measured an average signing-time of 0.54 ms, and verification time of 0.02 ms. RSA key generation has less consistent performance due to the rejection-sampling in prime number generation. We measured a median time of 51 ms with 31 ms and 83 ms as 0.25 and 0.75 quantiles and an average of 63 ms.

Pebbling H and the static key will involve up to $\frac{20}{2} = 10$ evaluations of H and the key-evolution of the static scheme per step. Individual evaluations of H are completely negligible ($\ll 0.01$ ms) next to all this.

Hence, $\Sigma.\mathsf{sign}$ takes about 0.54 ms, and $\Sigma.\mathsf{verify}$ roughly 0.27 ms. The runtime of $\Sigma.\mathsf{gen}$ is essentially the 7.7 min for XMSS key generation. Hashing a short bitstring 2^{20} times with SHA256 takes 61 ms which is essentially all that is needed to set up the other pebbling-structures and easily falls within the runtime-variation of the former.

For $\Sigma.\mathsf{evolve}$ we get $2 \cdot 63\,\text{ms} + 4.23\,\text{ms} + 2 \cdot 10 \cdot 4.23\,\text{ms} \approx 215\,\text{ms}$. More important is the speed with which multiple evolutions can be performed in quick succession (this is e.g. relevant on devices that have not been used in a while). In this case the creation of the dynamic signing-key and the time-lock puzzle can be left out, giving us 88.8 ms per epoch; Notably though the pebbling can be trivially parallelized to up to $2 \cdot 10 = 20$ threads which on our 6-core CPU gives us $4.23\,\text{ms} \cdot \lceil \frac{20}{2 \cdot 6} \rceil + 4.23\,\text{ms} \approx 12.7\,\text{ms}$, which is equivalent 3.65 s of parallel computation per fast-forwarded day.

3.10. Further Techniques

Our epochal signatures are designed to work as drop-in replacement for regular signatures. Aside from the ease of the conversion of the protocol in question, this also has the advantage that they are usable in other settings than just chats, such as explicitly authenticated key-exchanges.

This generic approach comes at the cost of not being able to exploit the properties that many chat-schemes have. In this section we will present further techniques that can be used to improve deniability in specialized settings.

3.10.1. Three-Level Signatures

Consider the most basic version of epochal signatures, namely a two-level signature-chain for which the keys of the second, dynamic level get published after expiration. Naturally this scheme can be extended to three levels by adding a middle layer between the static and the dynamic level. In real-time settings this can allow for extremely short validity-durations of the second level: Once the verifier is convinced the key-pair of the dynamic layer belongs to its owner, there is no more need to verify the public key and the secret key of the middle layer can be published right away.

3.10.2. Dynamic Signature-Chains

Consider the relatively short validity of the dynamic level signatures and their use in a context in which all intended receivers see all created signatures. The first aspect means that the main disadvantage of stateful signatures is not really meaningful in this scenario. The second aspect means that we can use linear signature chains with one-time signatures where every signature signs the message and the next public-key. Normally this would be too inefficient, but in the specific circumstances that we consider here this is not a problem: Each receiving party just has to maintain a trusted, "current" public key per other party and update it whenever she verifies a message successfully.

The first big advantage of this change is that one-time-signatures can be more efficient then regular signatures, especially in a post-quantum setting.

The second one is that old private keys can be published immediately after all intended receivers received and successfully verified the new key. If the publication is performed as an active action and not bound to time, this can even provide some protection against some online-judges: Since only members of a group know whether a dynamic layer key is expired, forwarding messages to people not in the group does not provide proof that the keys are still valid.

3.10.3. Signing Commitments

If we use the above solution with signature-chains in combination with few-times signatures, which might be desirable for efficient post-quantum security, there is a small risk that re-authenticating the dynamic level could require more signatures than safely available: If a user $\mathcal{U}$ is a very active writer in multiple channels with a high number of users joining within an epoch, all these users need to verify $\mathcal{U}$'s latest public key; this means that they either

have to verify the entire signature-chain, which would require them to receive all sent messages or to get a fresh signature from the mid-level key.

Given all messages of the current epoch to a user may however be very undesirable for confidentiality; One way to resolve this is with the use of commitments. Instead of signing the messages themselves, $\mathcal{U}$ signs a perfectly hiding commitment on the message and publishes the signature together with the opening-information for that commitment. Since the commitment is perfectly hiding it can be given to everyone, including parties who joined late without revealing anything about the content of $\mathcal{U}$'s past communication. We point out that the signature still provides unforgeability, as the ability to present a fresh plaintext-opening-information-pair for the commitment would by definition break the binding-property of the commitment-scheme.

In our opinion this technique is probably only useful if the goal is to instantiate epochal signatures with purely hash-based signature-schemes for reliable post-quantum-security, which is one of the reasons why we did not look into it further.

3.10.4. MAC in the Middle

Consider the actions performed on the middle layer:

1. A user $\mathcal{U}$ signs $\widetilde{pk}$ and sends the signatures to the other users.
2. Those other parties verify the signatures.
3. $\mathcal{U}$ publishes the secret key for deniability.

It turns out that there is no reason why the last two steps cannot be swapped: As long as the verifiers receive the signature before the publication of the secret key, it remains convincing even if it is only verified later.

This allows for the following change: Instead of signing the public-key of the dynamic layer with a signature-scheme, a user can simply "sign" it with a MAC-scheme, share the public-key together with the MAC, wait for confirmation of all intended receivers that they received these values and share the MAC-key with a signature of the of the top level for it.

This change has two main advantages: Firstly MACs are usually a much more efficient primitive than signatures, especially in a post-quantum setting. Secondly it improves deniability since everyone who can verify the MAC is also capable of forging it.

While it is in principle possible to use this technique at the dynamic layer, we believe that it is not practical for chats to do so, as it would cause possibly

long delays between the time when a message is received and the time when it is verified. Outside of chats the efficiency-gain may however be a valid trade-off.

In summary we consider this a very interesting technique for the middle-layer in chat-protocols, though it is not suitable for a generic drop-in replacement for signatures.

Algorithm 3: Σ.evolve. The green operations (line-numbers ending on "s") are only part of the simplified version, the purple ones (line-numbers ending on "c") only of the space-efficient complete one.

1 **fun** Σ.evolve(sk)**:**
2s $\quad pk, sk_r, \widehat{sk}, e, \_, \_ := sk$
2c $\quad pk, (sk_{r,\text{new}}, sk_{r,\text{exp}}), (\widehat{SK}_{\text{new}}, \widehat{SK}_{\text{exp}}), e, \_, \_ := sk$
3 $\quad \widehat{pk}, t_0, \Delta t, E, V := pk$
4 $\quad e_{\text{new}} := e + 1;\ e_{\text{exp}} := e - V$
5 $\quad \mathsf{sleep\_until}(t_0 + e_{\text{new}} \cdot \Delta t)$
6s $\quad r_{\text{new}} := sk_r[e_{\text{new}}];\ r_{\text{exp}} := sk_r[e_{\text{exp}}]$
7s $\quad \widehat{sk}_{\text{new}} := \widehat{sk}[e_{\text{new}}];\ \widehat{sk}_{\text{exp}} := \widehat{sk}[e_{\text{exp}}]$
6c $\quad sk'_{r,\text{new}}, r_{\text{new}} := \mathsf{Pebble}\left(sk_{r,\text{new}}\right)$
7c $\quad sk'_{r,\text{exp}}, r_{\text{exp}} := \mathsf{Pebble}\left(sk_{r,\text{exp}}\right)$
8c $\quad \widehat{SK}'_{\text{new}}, \widehat{sk}_{\text{new}}, := \mathsf{Pebble}\left(\widehat{SK}_{\text{new}}\right)$
9c $\quad \widehat{SK}'_{\text{exp}}, \widehat{sk}_{\text{exp}}, := \mathsf{Pebble}\left(\widehat{SK}_{\text{exp}}\right)$
10 $\quad r_{\overline{pk}} := \mathsf{H}(r_{\text{new}}, \widehat{pk} || e_{\text{new}} || 1)$
11 $\quad \overline{pk}_{\text{new}}, \overline{sk}_{\text{new}} := \overline{\Sigma}.\mathsf{gen}\left(1^\lambda; r_{\overline{pk}}\right)$
12 $\quad r_{tl} := \mathsf{H}(r_{\text{new}}, \widehat{pk} || e_{\text{new}} || 2)$
13 $\quad tl := \text{TL}.\mathsf{lock}\left(1^\lambda,\ V \cdot \Delta t,\ r_{\text{new}} || \widehat{sk}_{\text{new}};\ r_{tl}\right)$
14 $\quad \widehat{sk}', \hat{\sigma} := \widehat{\Sigma}.\mathsf{sign}\left(\widehat{sk},\ \overline{pk} || e_{\text{new}} || r_{\text{exp}} || \widehat{sk}_{\text{exp}} || tl\right)$
15 $\quad pinfo_{e_{\text{new}}} := (\overline{pk}_{\text{new}}, e_{\text{new}}, r_{\text{exp}}, \widehat{sk}_{\text{exp}}, tl, \hat{\sigma})$
16s $\quad sk' := (sk_r, \widehat{pk}, \widehat{sk}', e_{\text{new}}, \overline{sk}, pinfo_{e_{\text{new}}})$
16c $\quad sk' := (pk, (sk'_{r,\text{new}},\ sk'_{r,\text{exp}}), (\widehat{SK}'_{\text{new}}, \widehat{SK}'_{\text{exp}}),\ e_{\text{new}},\ \overline{sk},\ pinfo_{e_{\text{new}}})$
17 $\quad$ **return** $pinfo_{e_{\text{new}}}, sk'$

Algorithm 4: Σ.sign

1 **fun** Σ.sign$(sk,\ m)$**:**
2 $\quad \_, \_, \_, \_, \overline{sk}, pinfo_e := sk$
3 $\quad \overline{\sigma} := \overline{\Sigma}.\mathsf{sign}\left(\overline{sk},\ pinfo_e || m\right)$
4 $\quad \sigma := (\overline{\sigma}, pinfo_e)$
5 $\quad$ **return** σ

Algorithm 5: Σ.verify

fun $\Sigma.\mathsf{verify}\,(pk,\, e,\, \sigma,\, m)$**:**
- $\widehat{pk}, t_0, \Delta t, E, V := pk$
- $\overline{\sigma}, pinfo_{e'} := \sigma$
- $\overline{pk}_{e'}, e', r_{e'-V}, \widehat{sk}_{e'-V}, tl_{e'}, \hat{\sigma} := pinfo_{e'}$
- **if** $e \leq 0 \vee e' \leq 0 \vee e' + V \leq e \vee e' > e \vee e > E$**:**
  - **return** 0
- $b_0 := \widehat{\Sigma}.\mathsf{verify}\left(\widehat{pk},\, \hat{\sigma},\, \overline{pk}_{e'} || e' || r_{e'-V} || \widehat{sk}_{e'-V} || tl_{e'}\right)$
- $b_1 := \overline{\Sigma}.\mathsf{verify}\left(\overline{pk},\, \sigma,\, pinfo_{e'} || m\right)$
- **return** $b_0 \wedge b_1$

Algorithm 6: The key-extractor

fun $\mathsf{extract\_sk}(pk, pinfo_e)$**:**
- $\widehat{pk} := pk$
- $\_, e, r_{e-V}, \widehat{sk}_{e-V}, \_, \_ := pinfo_e$
- **for** $i \in \{e-1, \ldots, 0\}$**:**
  - $\widehat{sk}_i := \widehat{\Sigma}.\mathsf{update}\left(\widehat{sk}_{i+1}\right)$
  - $r_i := \mathsf{H}(r_{i+1}, \widehat{pk} || i+1 || 0)$
- $\widehat{sk}^* := [\widehat{sk}_0, \ldots, \widehat{sk}_e]$
- $sk_r^* := [r_0, \ldots, r_e]$
- **return** $(sk_r^*, \widehat{pk}, \widehat{sk}^*, 0, \perp, \perp)$

Algorithm 7: Our simulator $\mathcal{S}$

fun $\mathcal{S}(pk, pinfo_e, ts)$**:**
- $sk^* := \mathsf{extract\_sk}(pk, pinfo_e)$
- $ts' := [\,]$
- $e' := 0$
- **for** $(e, m) \in ts$**:**
  - $e - e'$ **times** : $\_, sk^* := \Sigma.\mathsf{evolve}\,(sk^*)$
  - $ts' || = \Sigma.\mathsf{sign}\,(sk^*,\, m)$
- **return** ts'

Algorithm 8: update_group updates a group to a new partial group state and causes a full interaction.

```
1  fun update_group(st, gr', SK):
2      gr := st.G[gr'.ID_gr]
3      A ←$ G*_gr
4      il := []
5      for U ∈ G_gr' \ G_gr:
6          il || = (A, (Add, gr, U), time, hid)
7      for U ∈ G_gr \ G_gr':
8          il || = (U, (Leave, gr), time, hid)
9      for U ∈ G_gr ∩ G_gr':
10         il || = (U, (SndM, gr, ϵ), time, hid)
11     _, st' := exec (st, il)
12     return st'
```

Algorithm 9: The simulator $\mathcal{S}$ for Π^*.

```
1  fun S(PK, sim_il,st, PI_e):
2      SK* := {extract_sk(pk, pinfo_e) | (pk, pinfo_e) ∈ PI_e}
3      U := list of all users mentioned in sim_il,st
4      st := (U, ∅, PK, SK*, ∅)
5      ts := ϵ
6      for x ∈ sim_il,:
7          if x is instruction:
8              ts', st := exec (st, [x])
9              ts || = ts'
10         else:
11             st := update_group(st, x, SK, )
12     return ts
```

4. Post Quantum WireGuard

This chapter is based on the paper "Post-quantum WireGuard" [HNS+20], authored jointly with Andreas Hülsing, Kai-Chun Ning, Peter Schwabe, and Philip R. Zimmermann, a slightly shortened version of which was published at IEEE S&P 2021 [HNS+21].

4.1. Introduction

WireGuard is a VPN protocol presented by Donenfeld in [Don17]. It combines modern cryptographic primitives with a simple design derived from the Noise framework [Per], a very small codebase, and very high performance.

These properties are achieved partially because WireGuard is "cryptographically opinionated" [Don17]: instead of supporting multiple cipher suites, WireGuard fixes X25519 [Ber06][1] for elliptic-curve Diffie-Hellman key exchange, Blake2 [ANWW13] for hashing, and ChaCha20-Poly1305 [Ber05, Ber08, NL18] for authenticated encryption. Not only are those primitives known for their outstanding software performance, fixing those primitives eliminates the need for an algorithm-negotiation phase, which keeps the protocol simple and its codebase small, and avoids any potential negotiation attacks. Also, high performance is achieved by implementing the protocol in the Linux kernel space, which eliminates the need for moving data between user and kernel space.

In addition to its superior performance and small codebase, WireGuard was designed to provide security properties that are not supported by other VPN software, e.g., identity hiding, and DoS-attack mitigation. The security considerations that lead to the design of WireGuard are laid out in [Don17]. Donenfeld and Milner give a computer-verified proof of the protocol in the symbolic model in [DM18]. In [DP18] Dowling and Paterson present a computational proof of the WireGuard handshake with an additional key-confirmation message.

[1] For naming of X25519, see https://mailarchive.ietf.org/arch/msg/cfrg/-9LEdnzVrE5RORux3Oo_oDDRksU.

Given its properties it is thus not surprising to see that WireGuard is becoming increasingly popular. For example, CloudFlare is working on "BoringTun", a WireGuard-based userspace VPN solution written in Rust [Bor]. Torvalds called WireGuard's codebase a "work of art" compared to OpenVPN and IPsec and advocated for its inclusion in Linux [Tor18]. WireGuard later became part of the mainline Linux kernel (version 5.6).

As WireGuard aims to be the next-generation VPN protocol, it is natural to see that security against quantum attackers played a role in its design as well, albeit a small one. Specifically, it allows users to include a symmetric shared key into the handshake, which protects against an attacker who records handshake transcripts now and attacks them in the future with a quantum computer [Don17, Sec. V.B]. Post-quantum asymmetric schemes are explicitly declared as *"not practical for use here"* by Donenfeld and are thus not included in the handshake. Recently, Appelbaum, Martindale, and Wu took another look at post-quantum security of WireGuard and proposed a small tweak to the protocol that aims at protecting against pretty much the same future quantum attacker with recorded transcripts [AMW19], but without requiring a long-term secure pre-shared key. The tweak assumes that public keys are typically not actually known to the attacker. If this is the case, then transmitting the hash of the public key instead of the public key itself prevents a future quantum attacker from ever learning the public key and thus from computing the corresponding secret key.

4.1.1. Contributions

In this chapter we present PQ-WireGuard, a post-quantum variant of the classical WireGuard handshake. Unlike the mitigation techniques described above and unlike various earlier works aiming at transitioning protocols to post-quantum security, we do not only aim for *confidentiality* against quantum attackers, but target full post-quantum security *including authentication*. The main design goal of PQ-WireGuard is to stay as close as possible to the original WireGuard protocol in terms of security and performance characteristics, i.e., PQ-WireGuard should

- achieve all the security properties of WireGuard, but now also resist attacks using a large-scale quantum computer;
- make a concrete choice of high-security, efficient cryptographic primitives instead of including an algorithm negotiation phase;
- finish the handshake in just one round trip;

- fit each of the two handshake messages into just one unfragmented IPv6 packet of at most 1280 bytes; and
- achieve much higher computational performance than other VPN solutions such as IPsec or OpenVPN.

PQ-WireGuard manages to tick all these boxes and thus shows that the assessment from the original WireGuard paper stating that post-quantum security is "not practical for use here" is no longer correct.

From Diffie-Hellman to KEMs

The original WireGuard protocol is heavily based on (non-interactive) Diffie-Hellman key exchange, which is not easy to replace straight-forwardly with post-quantum primitives. As of 2021 the only somewhat practical post-quantum non-interactive key exchange was CSIDH [CLM+18], which was (and is) both very young and rather inefficient. Furthermore, the security of concrete CSIDH parameters remains subject of heavy debated [BS20, BLMP19, Pei20, Ber19a]. We therefore took a different approach and first transformed the WireGuard protocol to a version using only interactive key-encapsulation mechanisms (KEMs). This approach was based on the KEM-based authenticated key exchange described in [FSXY12].

Security

Security of WireGuard is supported by the symbolic proof of Donenfeld and Milner [DM18] and the computational proof by Dowling and Paterson [DP18]. The symbolic proof covers more security properties than the computational proof and is computer verified. However, a correct computational proof gives stronger security guarantees as the proof makes fewer idealizing assumptions. We adapt the computational proof to the case of PQ-WireGuard which establishes together with the adapted symbolic proof [HNS+21] the same level of security guarantees as WireGuard. On the way, we point out (and fix) a few small mistakes in the computational proof. In order to allow for a standalone proof of the handshake we add an explicit key confirmation message to the PQ-WireGuard handshake as suggested in [DP18].

A concrete instantiation

The generic KEM-based approach allows us in principle to use any post-quantum KEM submitted to the NIST post-quantum project as a proposal for

future standardization[2]. Now the main challenge becomes one of public-key and ciphertext sizes: WireGuard operates over UDP and the existing codebase assumes that all handshake messages fit into one unfragmented IPv6 packet. The reason for this requirement is that increasing the number of packets in a handshake would make the state machine of the protocol more complex and contradict WireGuard's aim for simplicity in both protocol design and codebase. Fragmenting and reassembling IPv6 packets comes with various issues. For example, a denial-of-service (DoS) attack can fill up the reassembly buffer with fragments of packets that are never completed. This is just one example of IP fragmentation attacks [Atl12]. To prevent such attacks, some firewalls drop fragmented IPv6 packets, so avoiding fragmentation ensures that the protocol remains robust against such firewall configurations.

IPv6 packets are guaranteed not to be fragmented as long as they do not exceed 1280 bytes [DH17]. With the IPv6 header occupying 40 bytes and the UDP header occupying 8 bytes, there are 1232 bytes left for the content of handshake messages. In both initiation message and response, those 1232 bytes need to fit several MACs and protocol-specific fields alongside a public key and a ciphertext (for the initiator's packet) respectively two ciphertexts (for the responder's packet). For some of the schemes proposed to NIST, this is not much of a problem. For example, compressed SIKE [JAC+19] uses only 331 bytes for the public key and 363 bytes for the ciphertext, even at the highest security level. However, SIKE is not exactly known for its high computational performance; for example, it is more than an order of magnitude slower than most lattice-based KEMs. Furthermore it was broken [CD22] after the original publication of this work.

PQ-WireGuard uses two KEMs, namely Classic McEliece [BCL+19] and a passively secure variant of Saber [DKRV18, DKRV19]. One advantage of this solution for actual applications is that most security properties are guaranteed by the Classic McEliece scheme, considered by many as the most conservative choice among all NIST candidates. Another advantage is the computational efficiency (see below). Finally, our approach allows us to give a concrete example of an application that

1. works extremely efficiently with Classic McEliece, a cryptosystem that is often discarded as "impractical" because of its large public keys; and

2. heavily benefits from the savings in public-key and ciphertext size that lattice-based KEMs can achieve if they do not aim for active security.

[2] See https://csrc.nist.gov/Projects/Post-Quantum-Cryptography.

The second point may be seen as new insight into the question whether or not KEMs which only provide passive security really offer any benefits for real-world applications, which was repeatedly raised by Bernstein on the NIST pqc-forum mailing list [Ber19b, Ber19c]. The parameters our proposal uses achieve the "AES-192-equivalent" security level (NIST level 3).

Performance evaluation

To evaluate the performance of PQ-WireGuard, we compare the handshake efficiency of PQ-WireGuard with that of WireGuard, the strongSwan implementation of IPsec, and OpenVPN. We show that a PQ-WireGuard handshake is less than 60% slower than a WireGuard handshake, is more than 5 times faster than an IPsec handshake using Curve25519, and more than 1000 times faster than an OpenVPN handshake.

4.1.2. Related Work

Related work can be grouped in four categories.

First, there is ongoing effort for post-quantum security in the Noise framework [Per] that the WireGuard handshake is based on. Currently this effort only covers "transitional post-quantum security" (i.e., no post-quantum authentication), which is achieved by combining ephemeral-ephemeral ECDH with a post-quantum KEM (currently NewHope-Simple [ADPS16]). Noise calls this approach hybrid forward secrecy (HFS); the details are described in [Per18]. As the WireGuard handshake is one of the more complex Noise key-exchange patterns, our work may also be seen as a first step towards fully post-quantum Noise.

Second, there is a large body of work on authenticated key exchange including works on generic KEM-based constructions. Most important for this work is the generic KEM-based approach by Fujioka, Suzuki, Xagawa, and Yoneyama [FSXY12] (which can be seen as a generalization of "Efficient one-round key exchange in the standard model" [BCGP08]). All currently considered actively secure post-quantum KEMs start in their construction from a passively secure encryption scheme and obtain active security through variants of the Fujisaki-Okamoto (FO) transform [FO99]. In [HKSU18], Hövelmanns, Kiltz, Schäge, and Unruh present a generic AKE construction that starts directly from passively secure encryption schemes and moves some of the FO machinery into the AKE construction. A somewhat similar idea of reconsidering the FO transform in the context of authenticated key exchange is presented by Xue, Lu, Li, Liang, and He in [XLL+18]. However, the primitive

they start from in their generic construction is what they call a "2-key KEM". Also more specialized, non-generic, constructions of post-quantum AKEs have been described in the literature. In [ZZD+15], Zhang, Zhang, Ding, Snook, and Dagdelen describe a lattice-based AKE (which, however, was later outperformed by instantiating a generic construction with the lattice-based KEM Kyber in [BDK+18, Sec. 5]). Isogeny-based constructions were presented by Longa in [Lon18], by Xu, Xue, Wang, Au, Liang, and Tian in [XXW+19], and by Fujioka, Takashima, Terada, and Yoneyama in [FTTY19].

Third, there have been additional efforts on proving security properties of WireGuard and more generally Noise. Most notably, in [LBB19], Lipp, Blanchet, and Bhargavan present a computer-verified proof of security of the WireGuard handshake in the computational model. The proof is in the ROM; a meaningful translation to PQ-WireGuard would require first moving this proof to the QROM or the standard model. In [DRS19], Dowling, Rösler, and Schwenk introduce a generalization of the ACCE model from [JKSS12] and prove 8 out of the fundamental 15 Noise AKE patterns secure in this generalized ACCE model; the IK pattern used by WireGuard is not one of those 8 patterns. In [KNB19], Kobeissi, Nicolas, and Bhargavan present "Noise Explorer", a tool that fully automatically proves certain security properties of Noise AKE patterns in the symbolic model using ProVerif [Bla]. Adapting Noise Explorer to support KEM-based AKE such as the one we use in this chapter would certainly be interesting, but for the concrete case of PQ-WireGuard would not provide any more insight than our adaptation of the Tamarin [MSCB13] proof.

Finally, there are proposals to upgrade other VPN solutions to post-quantum security. Specifically, we are aware of two independent efforts to migrate OpenVPN [Inc19] to post-quantum cryptography. One of these efforts is described in the master's thesis by de Vries, which adds transitional security to OpenVPN through the use of McEliece as additional key exchange [dV16]. The other effort is PQCrypto-VPN by Easterbrook, Kane, LaMacchia, Shumow, and Zaverucha at Microsoft Research [EKL+19]. We give a performance comparison between our proposal and PQCrypto-VPN in Section 4.6.

4.1.3. Organization of this chapter

Section 4.2 gives a brief summary of the cryptographic primitives involved in the WireGuard handshake and then reviews the full handshake. Section 4.3 introduces the abstract, KEM-based construction of the PQ-WireGuard handshake and Section 4.4 analyzes its security. Section 4.5 describes the instanti-

ation of PQ-WireGuard using McEliece and a passively secure version of Saber. Finally, Section 4.6 presents benchmark results for PQ-WireGuard.

4.2. Preliminaries

In the following we briefly discuss the security notion under which we analyze PQ-WireGuard. We then recall some cryptographic primitives used by WireGuard and PQ-WireGuard, and eventually provide a brief description of the WireGuard handshake protocol. We start the discussion of security notions with a brief discussion of post-quantum security.

4.2.1. A Note on Post-Quantum Security

Our proofs in the computational model analyze the post-quantum security of PQ-WireGuard. This requires definitions of post-quantum security. In our case, the security notions for pre- and post-quantum security only differ with regard to the computational model of the attackers. More precisely, pre-quantum security assumes adversaries to be conventional probabilistic algorithms. For post-quantum security we assume adversaries to be quantum algorithms (which are probabilistic by nature). All honest parties are assumed to be conventional probabilistic algorithms. Consequently, all communication in our models is classical. We provide all definitions below with respect to probabilistic polynomial time (PPT) adversaries. We then obtain the post-quantum version by allowing the adversaries to be quantum polynomial time (QPT) algorithms.

This treatment is possible as our computational proofs are in the so-called standard model, i.e., the proofs do not make use of idealized primitives, like random oracles or ideal ciphers which would require quantum-access to oracles.

Looking at the symbolic model, we are not aware of research that analyzes the implications of considering quantum adversaries against conventional cryptography. Consequently, for now our proofs in the symbolic model are only known to apply to the pre-quantum setting.

4.2.2. Security Properties and Corruption Patterns

Before discussing formal models, we give some intuition on the security that WireGuard was designed to achieve. WireGuard considers a setting where an

initiator I initiates a secure connection with a *responder* R. The WireGuard handshake aims to achieve the following security goals:

- *Session-Key Secrecy*: The established session key is pseudorandom, i.e., indistinguishable from a random bit string for everyone except the initiator and the responder.
- *Session-Key Uniqueness*: The established session key is, with overwhelming probability, never repeated.
- *Entity authentication*: Both, initiator and responder, know who they are talking to. Specifically, it is practically infeasible for a party to impersonate another party.
- *Identity Hiding*: The identities of initiator and responder are only revealed to each other.
- *DoS Mitigation*: By DoS mitigation we refer to the first message by the initiator being authenticated. This allows a responder to reject forged messages before performing any costly public-key operations.

These security goals should even be preserved under corruption of secrets. All parties have a static long-term secret (usually the secret key of a key-pair). Identity is defined as knowledge of a certain long-term secret. In addition, parties have ephemeral secrets (think of ephemeral keys but also the randomness used during the execution of the protocol[3]) which are only used in a single execution of the protocol and are erased afterwards. We consider these a party's secrets and assume that they may be corrupted independently by an adversary. In addition, every pair of parties may or may not have a preshared secret that can be corrupted by the adversary as well. This allows to define different corruption patterns. In general, we consider *maximal exposure (MEX) attacks* [Kra05, Sec. 3.3],[FS12],[FSXY12] allowing adversaries to corrupt arbitrary combinations of static and ephemeral secrets.

However, certain corruption patterns allow for trivial, unpreventable attacks against certain security goals. For example, if all long-term secret data is compromised, there is no way to protect against active adversaries. In the following we discuss under which corruption patterns which security goals should still be achieved, explicitly excluding such trivial attacks.

[3]Some definitions limit the meaning of ephemeral secrets to ephemeral key pairs. We use it to refer to all temporary secret data in a party's state, especially all used randomness. This turns out to be important when using KEMs.

- *Session-Key Secrecy.* The session key remains pseudorandom if either the parties share an uncorrupted pre-shared key or if each party has at least one uncorrupted secret. This notion implies *perfect forward secrecy (PFS)* (also known as pre-compromise security) where an adversary learns the victim's secrets at some point in time and tries to learn the session key of previous sessions. In the case of weak-PFS the adversary is limited to sessions in which it did not actively interfere before.

- *Session-Key Uniqueness.* Session-key uniqueness holds against passive adversaries that only observe secrets of corrupted parties but do not actively change them.

- *Entity Authentication.* The handshake provides entity authentication under arbitrary corruption except for two cases. Assume Eve wants to impersonate Alice towards Bob then there exist two trivial corruption patterns.

  If Eve corrupts Alice's long-term secrets and all pre-shared secrets between Alice and Bob, authentication cannot be achieved.

  In addition to that, the impersonation may succeed if all of Bob's secrets are compromised, that is if Eve knows Bob's long-term and ephemeral secrets as well as the pre-shared secret between Alice and Bob. While this is no trivial attack in the sense that it cannot be prevented, it is often excluded as it describes a setting where Eve has essentially full control over Bob's system. In this case there are more direct ways than breaking cryptography to convince Bob that he is talking to Alice.

  All other attacks against entity authentication are mitigated, including *key-compromise impersonation (KCI) attacks*, where Eve tries to impersonate Alice towards Bob, while knowing all of Bob's long-term secrets. It also covers *unknown-key-share (UKS) attacks* in which Eve tricks an honest party into believing that they are communicating with someone else than they actually do.

- *Identity Hiding.* The identity of the initiator and the responder are hidden as long as both long-term secrets and the initiator's ephemeral secrets are uncompromised. We note here, that we regard compromise of a party's long-term secret as a reveal of its identity.

- *DoS Mitigation.* DoS mitigation can be achieved against adversaries that do not corrupt a pair of long-term key and pre-shared secret.

4.2.3. Formal Security Model

We analyze PQ-WireGuard in the computational model[4], where Dowling and Paterson introduced the notion of eCK-PFS-PSK security [DP18]. It extends the eCK-PFS notion of Cremers and Feltz [CF15] by the treatment of pre-shared keys. The notion of eCK-PFS security in turn is a strengthening of the eCK security notion [LLM07] that integrates perfect forward secrecy. In terms of the informal description above, eCK-PFS-PSK covers session-key secrecy, including PFS, and all the authentication-related security goals. What is not covered are session-key uniqueness, identity hiding, and DoS mitigation.

Following what Dowling and Paterson did for WireGuard, we prove security of PQ-WireGuard in the computational model with respect to eCK-PFS-PSK. The only difference is that we allow adversaries to be quantum algorithms as discussed above. For a formal description of eCK-PFS-PSK see Appendix A.1.

4.2.4. The WireGuard handshake

We are now ready to review the handshake protocol of WireGuard. In Figure 4.1 we first give a high-level view of the handshake, largely following the description in [DP18]. The initiator and responder are identified by their long-term, *static* public keys $\texttt{spk}_i$ and $\texttt{spk}_r$ (with corresponding secret keys $\texttt{ssk}_i$ and $\texttt{ssk}_r$, respectively). Those key pairs are generated before the first handshake between two parties and WireGuard assumes that the public keys are exchanged in a secure way (guaranteeing at least authenticity) before the first handshake.

From a cryptographic point of view, and in particular for the context of this chapter, what is most interesting is how the values H_k, κ_k, and C_k in Figure 4.1 are computed. This is laid out in Table 4.1, again largely following the description in [DP18]. The values $\texttt{lbl}_1$, $\texttt{lbl}_2$, and $\texttt{lbl}_3$ are fixed strings (see [Don17, Sec. V.D]). The value *cookie* is most of the time just 16 zero bytes, except when the server is under load and is sending out so-called "cookie replies" as denial-of-service countermeasure; for details, see [Don17, Sec. V.D7]. Note that Figure 4.1 includes the first application-data packet from the initiator. The reason is that this packet also serves as key confirmation of the handshake. This dual purpose of the first data packet is the reason that WireGuard cannot be proven secure in a modular way (separating handshake from data transport) without modification. For details see [DP18].

[4] The original paper also performed an analysis in the symbolic model using Tamarin that we don't include here because it was first and foremost the work of Ning.

Table 4.1.: Computation of seed values, keys, and hashes through the WireGuard handshake. For values of k with two rows, the first row denotes computation on the initiator side and the second row the corresponding computation on the responder side.

k	seed C_k	key κ_k	hash H_k
1	$\mathsf{H}(\mathtt{lbl}_1)$	—	$\mathsf{H}(C_1 \parallel \mathtt{lbl}_2)$
2	$\mathsf{KDF}_1(C_1, epk_i)$	—	$\mathsf{H}(H_1 \parallel \mathtt{spk}_r)$
3	$\mathsf{KDF}_1(C_2, \mathsf{DH.Shared}(esk_i, \mathtt{spk}_r))$	$\mathsf{KDF}_2(C_2, \mathsf{DH.Shared}(esk_i, \mathtt{spk}_r))$	$\mathsf{H}(H_2 \parallel epk_i)$
	$\mathsf{KDF}_1(C_2, \mathsf{DH.Shared}(\mathtt{ssk}_r, epk_i))$	$\mathsf{KDF}_2(C_2, \mathsf{DH.Shared}(\mathtt{ssk}_r, epk_i))$	$\mathsf{H}(H_2 \parallel epk_i)$
4	$\mathsf{KDF}_1(C_3, \mathsf{DH.Shared}(\mathtt{ssk}_i, \mathtt{spk}_r))$	$\mathsf{KDF}_2(C_3, \mathsf{DH.Shared}(\mathtt{ssk}_i, \mathtt{spk}_r))$	$\mathsf{H}(H_3 \parallel \mathtt{ltk})$
	$\mathsf{KDF}_1(C_3, \mathsf{DH.Shared}(\mathtt{ssk}_r, \mathtt{spk}_i))$	$\mathsf{KDF}_2(C_3, \mathsf{DH.Shared}(\mathtt{ssk}_r, \mathtt{spk}_i))$	$\mathsf{H}(H_3 \parallel \mathtt{ltk})$
5	—	—	$\mathsf{H}(H_4 \parallel \mathtt{time})$
6	$\mathsf{KDF}_1(C_4, epk_r)$	—	$\mathsf{H}(H_5 \parallel epk_r)$
7	$\mathsf{KDF}_1(C_6, \mathsf{DH.Shared}(esk_i, epk_r))$	—	—
	$\mathsf{KDF}_1(C_6, \mathsf{DH.Shared}(esk_r, epk_i))$	—	—
8	$\mathsf{KDF}_1(C_7, \mathsf{DH.Shared}(\mathtt{ssk}_i, epk_r))$	—	—
	$\mathsf{KDF}_1(C_7, \mathsf{DH.Shared}(esk_r, \mathtt{spk}_i))$	—	—
9	$\mathsf{KDF}_1(C_8, psk)$	$\mathsf{KDF}_3(C_8, psk)$	$\mathsf{H}(H_6 \parallel \mathsf{KDF}_2(C_8, psk))$
10	—	—	$\mathsf{H}(H_9 \parallel \mathtt{zero})$

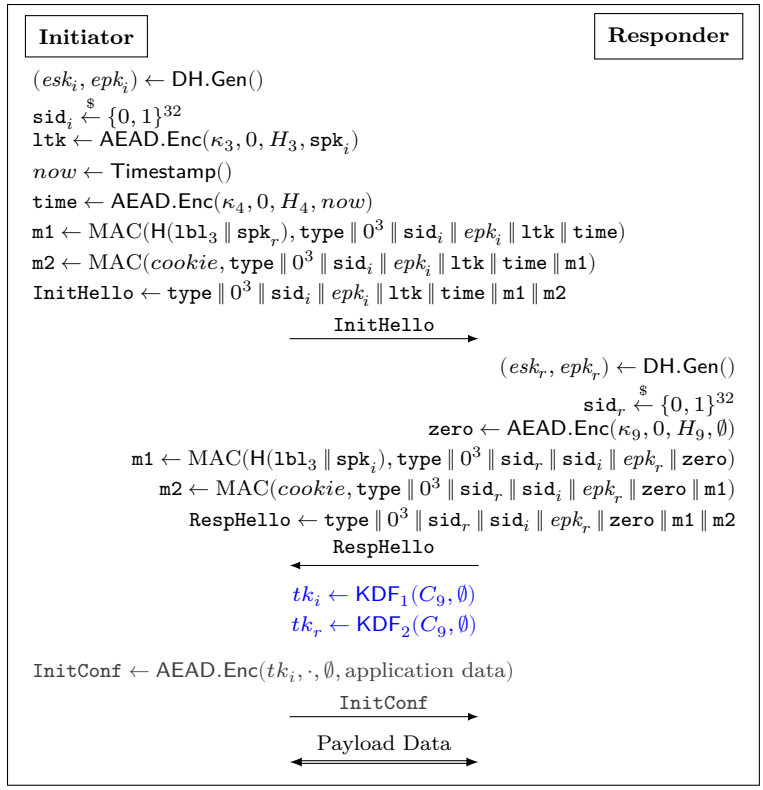

Figure 4.1.: High-level view on the WireGuard handshake.

4.3. From WireGuard to PQ-WireGuard

As outlined in Sections 4.1 and 4.2, the WireGuard handshake is heavily based on DH, which does not have an efficient and well-established post-quantum equivalent. Hence, in this section we describe how we replace DH by KEMs, for which well-established, efficient post-quantum instantiations exist. We start by considering a simplified view on the core of the DH-based WireGuard handshake.

In this simplified view, the initiator has a long-term static DH key pair $(\mathtt{ssk}_i, \mathtt{spk}_i)$ and the responder has a long-term static DH key pair $(\mathtt{ssk}_r, \mathtt{spk}_r)$. The handshake proceeds as depicted in Figure 4.2.

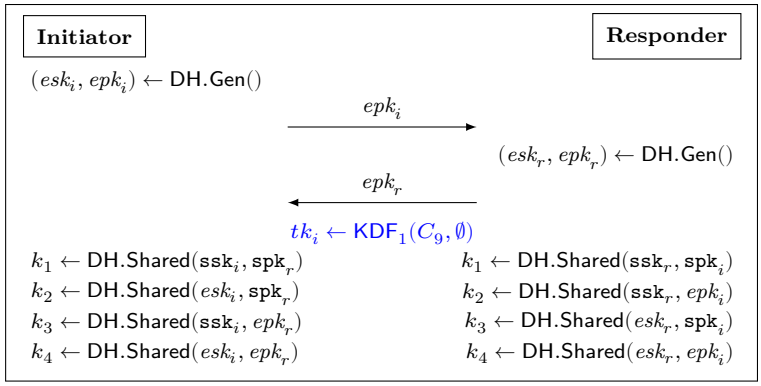

Figure 4.2.: The asymmetric core of the classical WireGuard-handshake.

The final session key is computed using the keys k_1, k_2, k_3, and k_4.

4.3.1. Moving from DH to KEMs

In [FSXY12], Fujioka, Suzuki, Xagawa, and Yoneyama describe an approach to authenticated key exchange using only KEMs; we largely follow their approach in our design. Towards our final proposal, let us first try to straightforwardly translate the DH-based approach to a KEM-based approach. The problem is, as described in Chapter 2.2, that we cannot perform a non-interactive key exchange, i.e., we cannot build an equivalent to the static-static DH computation of k_1. Let us for the moment ignore the static-static DH and start by translating the remainder of the handshake to a KEM-based handshake. For this, we will use an IND-CCA-secure KEM $\mathsf{CCAKEM} = (\mathsf{CCAKEM.Gen}, \mathsf{CCAKEM.Enc}, \mathsf{CCAKEM.Dec})$ and an IND-CPA-secure KEM $\mathsf{CPAKEM} = (\mathsf{CPAKEM.Gen}, \mathsf{CPAKEM.Enc}, \mathsf{CPAKEM.Dec})$. The initiator has a long-term static CCAKEM key pair $(\mathtt{ssk}_i, \mathtt{spk}_i)$ and the responder has a long-term static CCAKEM key pair $(\mathtt{ssk}_r, \mathtt{spk}_r)$. Now, the handshake proceeds as depicted in Figure 4.3.

The role of static-static DH.

This naive approach already preserves many of the security properties of the WireGuard handshake, but it lacks three properties that are achieved through the inclusion of the static-static DH.

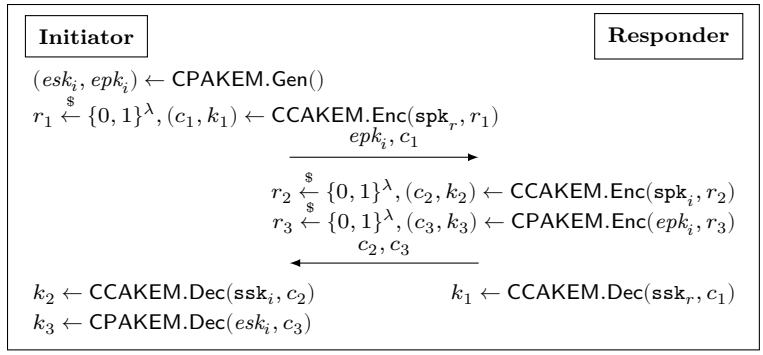

Figure 4.3.: The asymmetric core of the KEM-based PQ-WireGuard-handshake.

1. *Security under MEX attacks.* One corruption pattern in MEX attacks reveals all ephemeral secrets to the adversary, including the used randomness. The motivation for this pattern is a situation in which the protocol is executed on a device with a subverted, or simply broken RNG – in this case security can only be derived from the long-term secrets that have (ideally) been generated in a secure environment. In the above naive approach, we do not obtain any security in this scenario. The reason is that the randomness used by CCAKEM.Enc is corrupted and consequently, an adversary can recompute the shared secret simply running CCAKEM.Enc.

   One possibility to address this issue is to securely combine ephemeral randomness r with some long-term secret σ before using it as protocol input. In [FSXY15] this is done using $\mathsf{PRF}(r, \sigma) \oplus \mathsf{PRF}(\sigma', r')$ for two independent ephemeral values r and r' and two independent long-term secret values σ and σ', where $\oplus$ denotes exclusive or. This "twisted PRF" trick ensures that nothing beyond PRF security is required to prove this approach secure in the standard model. In the case of WireGuard we will see that we require a $\mathsf{dual-prf}$ assumption on KDF_1 anyway, so we can use this assumption here as well and simplify the construction to $\mathsf{KDF}_1(\sigma, r)$.

2. *Resistance to unknown-keyshare attacks.* The static-static DH is also the only line of defense in WireGuard against unknown-keyshare attacks. This is because the IDs (or public keys) of the two parties are not hashed

into the final session key. As briefly discussed in Section 5.1, WireGuard has the option to hash a pre-shared key psk into the final session key; by default psk is set to the all-zero string. In PQ-WireGuard we instead set psk to $\mathsf{H}(\mathtt{spk}_i \oplus \mathtt{spk}_r)$. This ensures that session keys are linked to the static public keys of the communicating parties and thus prevents unknown-keyshare attacks.

3. *Authenticated initiation.* Finally, the static-static DH ensures that the first message from the initiator is already authenticated. This allows the server to detect illegitimate messages at this very early stage and consequently abort the handshake. This is not a security property in the cryptographic sense but helps mitigate easy DoS attacks. If we follow the argumentation of [AMW19] stating that static public keys of WireGuard users are typically not public and hence not known to attackers, the same level of DoS protection is achieved by the default value of $psk = \mathsf{H}(\mathtt{spk}_i \oplus \mathtt{spk}_r)$. Users who do not want to rely on this assumption need to set psk to a secret shared key that is agreed on out-of-band to achieve the same level of DoS protection as in WireGuard.

Adding key confirmation.

As mentioned above, WireGuard uses the first application-data packet from the initiator for implicit key confirmation, which makes it impossible to prove the WireGuard handshake secure in the eCK-PFS-PSK model. The proof in [DP18], just like our computational proof, requires an explicit and separate `InitConf` key-confirmation message from the initiator at the end of the handshake. In PQ-WireGuard we add this explicit key-confirmation.

Putting it together.

Our full proposal for the KEM-based PQ-WireGuard handshake is given in Figure 4.4 and Table 4.2. Aside from translating all DH key exchanges, except for the static-static one, to corresponding KEM operations, we introduce the following changes to the WireGuard handshake:

- We use calls to $\mathsf{KDF}_1(\sigma_i, r_i)$ and $\mathsf{KDF}_1(\sigma_r, r_r)$ in steps 4 and 13 of Figure 4.4 to securely mix ephemeral randomness with long-term randomness. This is precisely the countermeasure against MEX attacks discussed above.

- We use $\mathsf{H}(\mathtt{spk}_i \oplus \mathtt{spk}_r)$ as default value for psk.

Table 4.2.: Computation of seed values, keys, and hashes through the PQ-WireGuard handshake. For values of k with two rows, the first row denotes computation on the initiator side and the second row the corresponding computation on the responder side. Highlighted in blue are differences to Table 4.1.

k	seed C_k	key κ_k	hash H_k
1	$\mathsf{H}(\texttt{lbl}_1)$	—	$\mathsf{H}(C_1 \parallel \texttt{lbl}_2)$
2	$\mathsf{KDF}_1(C_1, epk_i)$	—	$\mathsf{H}(H_1 \parallel \texttt{spk}_r)$
3	$\mathsf{KDF}_1(C_2, \texttt{shk1})$	$\mathsf{KDF}_2(C_2, \texttt{shk1})$	$\mathsf{H}(H_2 \parallel epk_i)$
	$\mathsf{KDF}_1(C_2, \mathsf{CCAKEM.Dec}(\texttt{ssk}_r, \texttt{ct1}))$	$\mathsf{KDF}_2(C_2, \mathsf{CCAKEM.Dec}(\texttt{ssk}_r, \texttt{ct1}))$	$\mathsf{H}(H_2 \parallel epk_i)$
4	$\mathsf{KDF}_1(C_3, psk)$	$\mathsf{KDF}_2(C_3, psk)$	$\mathsf{H}(H_3 \parallel \texttt{ltk})$
5	—	—	$\mathsf{H}(H_4 \parallel \texttt{time})$
6	$\mathsf{KDF}_1(C_4, \texttt{ct2})$	—	$\mathsf{H}(H_5 \parallel \texttt{ct2})$
7	$\mathsf{KDF}_1(C_6, \mathsf{CPAKEM.Dec}(esk_i, \texttt{ct2}))$	—	—
	$\mathsf{KDF}_1(C_6, \texttt{shk2})$	—	—
8	$\mathsf{KDF}_1(C_7, \mathsf{CCAKEM.Dec}(\texttt{ssk}_i, \texttt{ct3}))$	—	—
	$\mathsf{KDF}_1(C_7, \texttt{shk3})$	—	—
9	$\mathsf{KDF}_1(C_8, psk)$	$\mathsf{KDF}_3(C_8, psk)$	$\mathsf{H}(H_6 \parallel \mathsf{KDF}_2(C_8, psk))$
10	$\mathsf{KDF}_1(C_9, \emptyset)$	$\mathsf{KDF}_2(C_9, \emptyset)$	$\mathsf{H}(H_9 \parallel \texttt{zero})$

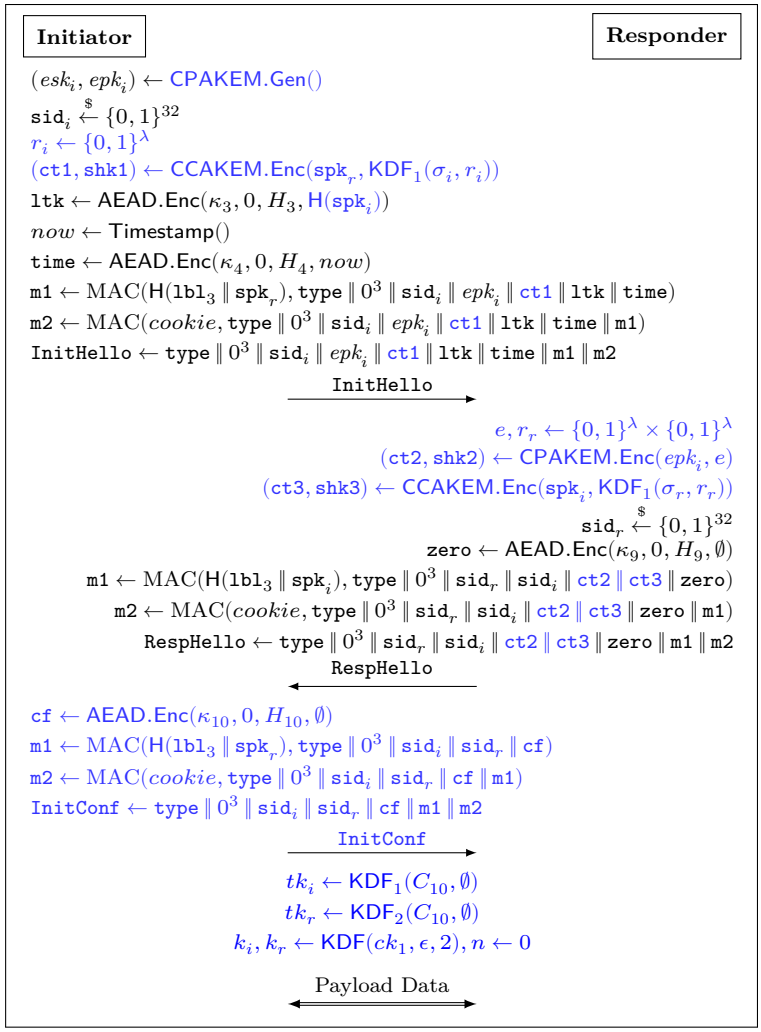

Figure 4.4.: High-level view on our PQ-WireGuard handshake. Highlighted in blue are differences to Figure. 4.1.

- Instead of feeding $\mathtt{spk}_i$ into AEAD.Enc in step 5, we use $\mathsf{H}(\mathtt{spk}_i)$. This is essentially the same trick proposed in [AMW19], except that we need it

for a very different reason. In [AMW19] the reason is to add some protection against future quantum attackers who are recording handshakes today. For us the reason is simply a size reduction from the potentially large public key of CCAKEM to a 32-byte hash of this public key.

- We add the explicit key-confirmation message `InitConf`.

4.4. Security analysis

We prove the security for PQ-WireGuard in the computational model. Combined with the symbolic proof in the original publication we thereby provide the same level of security guarantees as for WireGuard.

In the computational model we prove that the PQ-WireGuard handshake, like the WireGuard handshake (with added key confirmation), manages to achieveeCK-PFS-PSK-security. While certainly on the stronger end of security notions for authenticated key-exchange, eCK-PFS-PSK only proves session-key secrecy and authenticity properties. Further notions that PQ-WireGuard also targets, like identity hiding and DoS-mitigation, are not covered by it.

These additional notions are covered by the symbolic proof. The symbolic proof not only covers additional security properties but also has the advantage of being computer-verified. However, this comes at the cost of being done in the symbolic model which treats all building blocks as ideal and consequently can only provide a heuristic argument.

4.4.1. The Computational Proof

To prove that the PQ-WireGuard handshake achieves eCK-PFS-PSK-security, we adapt the computational proof for WireGuard [DP18] by Dowling and Paterson (who kindly provided us with their LaTeX-sources) to PQ-WireGuard. The core of this adaptation is to replace proof steps (i.e., game-hops) making use of either the PRF-ODH- or the DDH-assumptions by generic KEM-security- and prf-assumptions. Most of these changes are straightforward and readers who are familiar with the original proof should find the result familiar.

On a high level both proofs consist of the same case-distinction between whether the adversary tries to impersonate a party or learn information about the established key and the ways in which the adversary is allowed to corrupt parties. For each case the proof uses a sequence of games to show that the adversary has to either directly break the authenticity of the AEAD-scheme for a successful impersonation attack or distinguish two information-

theoretically indistinguishable bit strings to learn any non-trivial information about the key.

The majority of game hops are ones where the prf or the $\mathsf{prf}^{\mathsf{swap}}$ assumptions are used. In these game-hops the output of KDF, used to combine two intermediate values, at least one of which is random (which one depends on the adversarial corruption), gets replaced by a random value. These "symmetric game hops" are essentially the same in the WireGuard and the PQ-WireGuard proof.

The other major category of game hops is those where the output of some asymmetric primitive is replaced by a random value. For WireGuard, these are the cases where two DH shares get combined and hashed afterwards. In this case, different versions of the PRF-ODH assumption are used to argue indistinguishability of the games before and after the hop. For PQ-WireGuard, these steps use KEM encapsulations and decapsulations. In these cases, indistinguishability can be argued using the IND-CPA security of CPAKEM and the IND-CCA security of CCAKEM.

The differences between the proofs for WireGuard and PQ-WireGuard are not just limited to these asymmetric game hops: The ways values are combined in some cases in PQ-WireGuard differ substantially from WireGuard. This is necessary to deal with the more limited abilities of KEMs when compared to Diffie-Hellman. As a consequence, we had to add multiple new symmetric game hops, particularly around most asymmetric game hops.

In addition to that we noticed one minor mistake in the WireGuard proof that also directly affects our proof. The WireGuard proof claims that it is sufficient for KDF to be a prf. This turns out to be too weak. KDF is used to combine two inputs. While in different corruption settings there is always one input that is indistinguishable from random for the attacker, but it is not always the same input. Consequently, the function actually has to be a dual-PRF (which can be keyed on either input). For the most part this occurs in asymmetric game hops where the prf-assumption is "hidden" in the PRF-ODH assumption, but it also occurs in one symmetric hop. We notified the authors of the WireGuard proof who acknowledged the issue.

Given these changes, we are able to show that there is no efficient adversary against the eCK-PFS-PSK security of PQ-WireGuard under the assumptions that the used KDF is a secure dual-PRF, that the used KEMs are respectively IND-CCA and IND-CPA secure, that the IND-CCA-secure KEM has perfect correctness, and that the used AEAD scheme is secure in terms of authenticity. More specifically, we show that for every (possibly quantum) adversary $\mathcal{A}$ we can construct a set $\mathcal{R}$ of (possibly quantum) reduction algorithms $\mathcal{R}_i$

that use oracle access to $\mathcal{A}$ to break one of the security assumptions running in about the same time as $\mathcal{A}$ so that:

$$\begin{aligned}
&\mathsf{Adv}^{\mathsf{eCK\text{-}PFS\text{-}PSK}}_{\mathsf{pqWG},\mathsf{clean}_{\mathsf{eCK\text{-}PFS\text{-}PSK}},n_P,n_S,\mathcal{A}}(\lambda) \\
&\leq n_P^2 n_S \left(\begin{array}{rrcl}
 & (7n_S+9) & \cdot & \mathsf{Adv}^{\mathsf{prf}}_{\mathsf{KDF},\mathcal{R}}(\lambda) \\
+ & (2n_S+4) & \cdot & \mathsf{Adv}^{\mathsf{prf}^{\mathsf{swap}}}_{\mathsf{KDF},\mathcal{R}}(\lambda) \\
+ & (n_S+2) & \cdot & \mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathsf{CCAKEM},\mathcal{R}}(\lambda) \\
+ & n_S & \cdot & \mathsf{CPAKEM}.\delta \\
+ & n_S & \cdot & \mathsf{Adv}^{\mathsf{IND\text{-}CPA}}_{\mathsf{CPAKEM},\mathcal{R}}(\lambda) \\
+ & 2 & \cdot & \mathsf{Adv}^{\mathsf{auth\text{-}aead}}_{\mathsf{AEAD},\mathcal{R}}(\lambda) \\
+ & (n_S+2) & \cdot & \frac{n_s}{2^\lambda}
\end{array} \right)
\end{aligned}$$

where n_P is the number of parties and n_S is the number of sessions. Here we use $\mathsf{Adv}^{\mathsf{prop}}_{\mathsf{F},\mathcal{R}}(\lambda)$ for the maximum success probability over all $\mathcal{R}_i \in \mathcal{R}$ against property prop of building block F. For the formal security proof and a slightly tighter and more precise bound see Section 4.7.

Finally we would like to point out a pleasant side-result of the strong security-notion and the use of two different KEMs that correspond to static and ephemeral keys: If we model the break of a KEM as the reveal of all secret keys (and therefore also encapsulated secrets) then a break of either KEM does not break the confidentiality of PQ-WireGuard as long as there is no further corruption:

A break of our CCA-KEM would be equivalent to a corruption of all static secrets, but notably not the ephemeral keys used with the CPA-KEM. As long as no ephemeral secrets are compromised, eCK-PFS-PSK-security still promises that the established key remains confidential. (However, authenticity is trivially broken.)

A break of our CPA-KEM on the other hand would be equivalent to a corruption of all ephemeral secrets, but not of the static secrets that are used with the CCA-KEM. As long as no static secrets are compromised eCK-PFS-PSK-security still promises authenticity and confidentiality, losing PFS though.

The consequence of this is an increased robustness of the scheme which is relevant to us as most post-quantum primitives (in case of our proposed instantiation particularly the upcoming modified version of Saber) are rather new and therefore more likely to break than more traditional schemes. The practical impact is that PQ-WireGuard already provides some of the properties of hybrid protocols that aim at redundancy of security assumptions

by combining cryptographic schemes that use different security assumptions (e.g., ECC and lattice-based schemes).

Identity Hiding.

Identity hiding is a property that was proven in the symbolic proof for WireGuard. When adapting the proof to the KEM setting of PQ-WireGuard Tamarin successfully proves the property. However, it turns out the model in this case makes an idealizing assumption that is not necessarily true in the KEM setting: The Tamarin model assumes a key hiding property, i.e., that a key encapsulation (in case of DH a key exchange message) does not allow to learn under which public key it was produced. For DH key exchange this can be derived from the DDH assumption (against passive adversaries) or the PRF-ODH assumption (against active adversaries).

For KEMs this assumption is not implied by the standard security assumptions of IND-CPA and IND-CCA security. For a counter example consider a KEM that makes the public key part of a key ciphertext: This would not invalidate IND-CPA or IND-CCA security, but it would trivially reveal the identity of the receiver. What is formally required to justify the applicability of the model is a KEM version of the notion of indistinguishability of keys (IK) [BBDP01]. For active adversaries IK-CCA is required, for passive adversaries IK-CPA is sufficient. In [YMT17] it is shown that the public key encryption scheme underlying Classic McEliece achieves IK-CPA security. Based on this we conjecture that the Classic McEliece KEM achieves IK-CCA security, but we do not formally prove it.

4.5. Instantiation with McEliece and Saber

The generic approach for a purely KEM-based variant of WireGuard allows us in principle to instantiate the protocol with any post-quantum KEM(s) with the required security properties. In this section we describe the concrete instantiation we chose. We selected the Classic McEliece [BCL+19] IND-CCA KEM and an IND-CPA secure variant of Saber [DKRV19, DKRV18]. One could say that this choice—just like the choices of primitives in WireGuard—is "cryptographically opinionated". The criteria by which we made this choice are the following:

- stick to primitives that are in the second round of the NIST PQC project and thus have potential to become a future standard;

- choose parameters that reach NIST security level 3 ([fSaT16, Sec. 4.A.5]);
- do not increase the number of required unfragmented IPv6 packets for the handshake compared to WireGuard (one sent by the initiator and one by the responder, plus the key-confirmation by the initiator that is implicit through application-data transmission in WireGuard and explicit in PQ-WireGuard).
- pick primitives that have high-performance timing-attack protected implementations;
- pick "conservative" primitives, i.e, primitives building on a history of cryptanalytic results;
- stay away from primitives that the submitters declare to be encumbered by patents; and
- do not modify or tweak primitives in any way that would invalidate security reductions.

The most limiting of these criteria is to fit the initiator's and the responder's handshake messages into one IPv6 packet each. IPv6 mandates every link in the internet to support an MTU of at least 1280 bytes [DH17, Sec. 5]. Out of those 1280 bytes, 40 are required for the IPv6 header and another 8 are required for the UDP header. This leaves 1232 bytes for the WireGuard handshake payloads. In the initiator's message, the fields `type`, 0^3, $\texttt{sid}_i$, `ltk`, `time`, `m1`, and `m2` together occupy 116 bytes, which leaves 1116 bytes for a CPAKEM public key and a CCAKEM ciphertext. In the responder's message, the fields `type`, 0^3, $\texttt{sid}_i$, $\texttt{sid}_r$, `zero`, `m1`, and `m2` together occupy 60 bytes, which leaves 1172 bytes for a CPAKEM ciphertext and a CCAKEM ciphertext.

Classic McEliece as CCAKEM.

Note in Figure 4.4 that the handshake never sends public keys of CCAKEM, and that the computation does not involve any CCAKEM.Gen operations. This means that for the instantiation of CCAKEM we are mainly concerned about ciphertext size with secondary criteria being encapsulation and decapsulation speed. Out of all round-2 NIST PQC candidate KEMs[5], Classic McEliece has the smallest ciphertext by far, weighing in at only 188 bytes for the level-3 parameter set `mceliece460896`. Also, Classic McEliece comes

[5] For an overview, see https://pqc-wiki.fau.edu/

with very fast timing-attack-protected software for encapsulation and decapsulation, which makes it the ideal choice of primitive for our use case. Note that McEliece is often regarded as a conservative, but rather inefficient choice, because of its slow key generation and large public keys – however, these disadvantages are precisely the aspects that do not matter for us here.

Tweaked Saber as CPAKEM.

With the rather straightforward choice of Classic McEliece as instantiation of CCAKEM fixed, we need to find an IND-CPA KEM among the NIST candidates that has public keys of at most 928 bytes and ciphertexts of at most 984 bytes for parameters that reach the NIST security level 3. The only KEMs that meet these criteria are Round5 [BBF+19], SIKE [JAC+19], and ROLLO-I [MAB+19]. Unfortunately, none of these three meets our other criteria. Round-5 is covered by patents held by the submitters; SIKE is rather slow, for example more than an order of magnitude slower than most lattice-based KEMs, and ROLLO-I cannot be seen as a particularly conservative choice. Specifically, in the document explaining the choice of round-2 candidates [AASA+19], NIST writes about the rank-based candidate ROLLO-I:

> *"Nonetheless rank-based cryptography is quite new and not as well studied as lattice-based cryptography or code-based cryptography using the Hamming metric. More cryptanalysis on rank-based primitives would be valuable."*

However, among the remaining candidates, there are multiple lattice-based KEMs with public keys and ciphertext that are only slightly larger than what we need. Also, most of them aim for IND-CCA security (which we do *not* need to instantiate CPAKEM) and some of them allow to reduce the size of public keys and ciphertexts at the expense of achieving only IND-CPA security and increasing failure probability.

Concretely, Saber already includes public-key and ciphertext compression, and, in order to achieve IND-CCA security, carefully chooses parameters to minimize sizes while keeping the failure probability δ cryptographically negligible. We decided to propose an IND-CPA version of Saber, which compresses public keys and ciphertexts even further. This comes at the additional advantage that the underlying hard lattice problem becomes harder, but at the expense of significantly increased failure probability.

By tweaking some parameters of Saber we designed a variant that we will henceforth call "Dagger" [HNS+21]. Compared to Saber, the modifications in Dagger reduce the public-key size from 992 bytes to 896 bytes and the

ciphertext size from 1088 bytes to 960 bytes, which is well within our limits. In addition to the modified parameters Dagger does not use the Fujisaki-Okamoto transform [FO99], i.e., the construction that Saber uses to build an IND-CCA KEM from an IND-CPA public-key encryption scheme.

4.6. Performance analysis

In this Section, we summarize the performance benchmarks [HNS+21] of PQ-WireGuard that compared it to original WireGuard (version 0.0.20191206), IPsec (strongSwan in version U5.6.2/K4.15.0-72-generic), OpenVPN (version 2.4.4, linked against OpenSSL 1.1.1), OpenVPN-NL (version 2.4.7, linked against mbed TLS 2.16.2), and PQCrypto-VPN (OpenVPN 2.4.4, linked against OQS-OpenSSL 1.0.2 [MS20]). OpenVPN-NL is a branch of OpenVPN, which is mandated for critical infrastructure in the Netherlands by the Dutch government, while PQCrypto-VPN is the aforementioned VPN software from Microsoft [EKL+19] based on OpenVPN and the Open Quantum Safe (OQS) framework [MS20]. The measurements for WireGuard include the first application-data packet (the benchmark used zero-length application data), which also serves as key confirmation. In other words, the handshake was considered finished on the responder (server) side only at the point when the server was ready to send application data.

The used PQ-WireGuard software was based on the original WireGuard implementation. The used implementation for Classic McEliece was the "`avx2`" software targeting recent 64-bit Intel processors, which has been submitted to SUPERCOP [BL] by the Classic McEliece team. The implementation of Dagger was based on the Saber reference implementation.

The experiments were carried out between two virtual machines managed by VMware's "vSphere" in version 6.7 and connected through a virtual Ethernet link (VMware "vSwitch") with a bandwidth limit of 10 Gbit/s. Both virtual machines are running Linux kernel 4.15.0. The underlying physical machine is powered by Intel Xeon Gold 6130 (Skylake) CPUs running at 2.1 GHz.

We compare the handshake efficiency by the following metrics: the amount of traffic, the number of packets exchanged, and the time span of the handshake. The client time span is the elapsed time between when the client starts any computation for a handshake and when session keys are derived from the handshake on the client side. Similarly, the server time span is the elapsed time between the server receiving an initiation packet from the client and the server being ready to send application data to the client.

The handshake protocol of each VPN software was invoked 1000 times to compute the average and standard deviation (enclosed by parentheses) of those metrics. The results with IPv4 and IPv6 are presented in Table 4.3 and Table 4.4, respectively. In both tables, the amount of traffic includes the 14-byte Ethernet frame headers.

Table 4.3.: Resource Requirements for VPN handshake protocols over IPv4, numbers in parentheses are standard deviation. [HNS+21]

VPN Software	Packet Number	Traffic (bytes)	Client Time (milliseconds)	Server Time (milliseconds)
WireGuard	3 (0)	398 (0)	0.584 (0.508)	0.494 (0.507)
PQ-WireGuard (this work)	3 (0)	2594 (0)	0.975 (0.442)	0.745 (0.245)
IPsec (RSA-2048)	6 (0)	4123 (0)	17.046 (0.826)	11.823 (0.726)
IPsec (Curve25519)	4 (0)	2145 (0)	5.127 (0.375)	2.807 (0.431)
OpenVPN (RSA-2048)	21.005 (0.071)	7535.507 (7.940)	1150.872 (244.288)	1144.994 (251.304)
OpenVPN (NIST P-256)	19.005 (0.007)	5408.572 (7.997)	1152.238 (242.014)	1150.310 (253.582)
OpenVPN-NL (RSA-2048)	19.005 (0.007)	5685.585 (8.155)	1157.732 (244.015)	1151.446 (246.534)
OpenVPN-NL (NIST P-256)	19.006 (0.078)	5681.711 (8.979)	1159.099 (241.534)	1156.482 (235.703)
PQ-OpenVPN (Frodo-752)	63.001 (0.032)	34348.114 (3.569)	1151.529 (235.234)	1143.337 (238.465)
PQ-OpenVPN (SIDHp503)	23.003 (0.055)	8536.345 (6.188)	1266.838 (258.101)	1265.332 (264.271)

We see that both WireGuard and PQ-WireGuard only require 3 packets. We also see that in PQ-WireGuard, the total time required for the handshake on the client-side increases by less than 70% compared to WireGuard, at least when it is run over a high-speed network link as in our experiments. The time required for server-side computations increases by just over 50% compared to WireGuard. The computational effort for both WireGuard and PQ-WireGuard is dominated by public-key cryptography; we would expect that future improvements to the McEliece or Dagger software could bring PQ-WireGuard even closer to the performance of WireGuard.

Just as the original WireGuard software, PQ-WireGuard outperforms the main competitors IPsec and OpenVPN in terms of handshake time, computation time on the server, number of transmitted packets, and amount of transmitted data. Specifically, the PQ-WireGuard handshake is about 5 times faster than the handshake of IPsec and more than three orders of magnitude

Table 4.4.: Resource Requirements for VPN handshake protocols over IPv6, numbers in parentheses are standard deviation. [HNS+21]

VPN Software	Packet Number	Traffic (bytes)	Client Time (milliseconds)	Server Time (milliseconds)
WireGuard	3 (0)	458 (0)	0.592 (0.399)	0.480 (0.389)
PQ-WireGuard (this work)	3 (0)	2654 (0)	1.015 (0.618)	0.786 (0.621)
IPsec (RSA-2048)	6 (0)	4299 (0)	17.188 (0.712)	11.912 (0.535)
IPsec (Curve25519)	4 (0)	2281 (0)	5.226 (0.575)	2.822 (0.436)
OpenVPN (RSA-2048)	21.003 (0.055)	7955.409 (7.319)	1148.733 (250.513)	1142.650 (243.184)
OpenVPN (NIST P-256)	19.005 (0.007)	5788.610 (9.423)	1139.140 (247.659)	1133.944 (240.691)
OpenVPN-NL (RSA-2048)	19.005 (0.072)	6065.700 (9.665)	1162.649 (261.078)	1151.790 (246.363)
OpenVPN-NL (NIST P-256)	19.001 (0.003)	6061.138 (4.304)	1159.627 (252.989)	1153.949 (247.470)
PQ-OpenVPN (Frodo-752 [BCD+16])	63.006 (0.078)	35608.817 (10.324)	1160.922 (259.246)	1155.713 (245.614)
PQ-OpenVPN (SIDHp503)	23.005 (0.072)	8996.684 (9.449)	1277.172 (251.461)	1269.074 (257.427)

faster than the handshake of OpenVPN, all while offering full protection against future attackers equipped with large quantum computers.

4.7. Full Proof

In this section we present the full security-proof of our scheme. Most of it is taken verbatim from the WireGuard-proof[DP18] by Benjamin Dowling and Kenneth G. Paterson whom we thank for kindly providing us with their LaTeX-sources; the highlighted parts (such as this one) are our own modifications to that proof so that it also works with PQ-WireGuard. This was done with the intention of allowing readers who are already familiar with the older proof to concentrate on our changes to it. For the same reason we also tried to keep the style of our additions as close to the original paper as possible.

One thing that we would like to point out concerns the game-hops that use the prf- and the prf^{swap} assumptions: One might intuitively assume that keys or messages could potentially collide and result in a tightness-loss (like the ones that actually occur in *Game 5a* of **Case 1**, *Game 5a* of **Case 2** and *Game 3a* of **Case 3.5**). This is not the case however: Since the key is random and independent from any other key in all these cases, any potential

collision of the key is already part of the adversarial advantage against the $\mathsf{prf}/\mathsf{prf}^{\mathsf{swap}}$-security. As any (non-colliding) key is furthermore only used once (except for the aforementioned cases), there cannot be any colliding messages. The later part actually strengthens the real security of the protocol, since it massively reduces the kinds of attacks against the KDF that can be converted into attacks against PQ-WireGuard.

While Dowling and Paterson don't make this statement as explicit (possibly because they considered it obvious), it applies to their proof just as well.

An additional concern that the move to post-quantum-schemes introduces is that of imperfect correctness of our asymmetric building-block: While our CCAKEM (Classic McEliece) offers perfect correctness, the same does not hold for the CPAKEM (Dagger) that we use to achieve forward-secrecy and offers merely δ-correctness (that is, with a probability δ the decapsulation-algorithm may output a different key than the encapsulation algorithm or even no key at all!). This causes a small loss in tightness. We remark here that our analysis is specific to our instantiation of these primitives in that our proof assumes perfect correctness for CCAKEM and would thus require a slight modification before it could be adapted for merely δ-correct CCAKEM.

Theorem 11. *The modified WireGuard handshake protocol pqWG is eCK-PFS-PSK-secure with cleanness predicate* $\mathsf{clean}_{\mathit{eCK\text{-}PFS\text{-}PSK}}$ *(capturing forward secrecy and resilience to KCI attacks). That is, for any (potentially quantum) algorithm* $\mathcal{A}$ *against the eCK-PFS-PSK key-indistinguishability game (defined in Figure A.1) the adversarial advantage* $\mathsf{Adv}^{\mathit{eCK\text{-}PFS\text{-}PSK}}_{\mathit{pqWG},\mathsf{clean}_{\mathit{eCK\text{-}PFS\text{-}PSK}},n_P,n_S,\mathcal{A}}(\lambda)$ *is bounded by a polynomial factor of* $\mathcal{A}$*'s advantage in the* $\mathsf{dual}-\mathsf{prf}$*, IND-CCA,*

IND-CPA and auth-aead games. Specifically:

$$
\begin{aligned}
&\mathsf{Adv}^{\textit{eCK-PFS-PSK}}_{\textit{pqWG},\mathsf{clean}_{\textit{eCK-PFS-PSK}},n_P,n_S,\mathcal{A}}(\lambda) \\
&\leq\ n_P^2 \cdot n_S \left(\begin{array}{r} \frac{n_s}{2^\lambda} + \mathsf{Adv}^{\textit{IND-CCA}}_{\mathsf{CCAKEM},\mathcal{R}}(\lambda) \ +\ 6 \cdot \mathsf{Adv}^{\mathsf{prf}}_{\mathsf{KDF},\mathcal{R}}(\lambda) \\ +\ 2 \cdot \mathsf{Adv}^{\textit{prf}^{\textit{swap}}}_{\mathsf{KDF},\mathcal{R}}(\lambda) \ +\ \mathsf{Adv}^{\textit{auth-aead}}_{\mathsf{AEAD},\mathcal{R}}(\lambda) \end{array} \right) \\
&+\ n_P^2 \cdot n_S \left(\begin{array}{r} \frac{n_s}{2^\lambda} + \mathsf{Adv}^{\textit{IND-CCA}}_{\mathsf{CCAKEM},\mathcal{R}}(\lambda) \ +\ 3 \cdot \mathsf{Adv}^{\mathsf{prf}}_{\mathsf{KDF},\mathcal{R}}(\lambda) \\ +\ 2 \cdot \mathsf{Adv}^{\textit{prf}^{\textit{swap}}}_{\mathsf{KDF},\mathcal{R}}(\lambda) \ +\ \mathsf{Adv}^{\textit{auth-aead}}_{\mathsf{AEAD},\mathcal{R}}(\lambda) \end{array} \right) \\
&+\ \max \left\{ \begin{array}{l}
n_P^2 \cdot n_S^2 \left(\begin{array}{l} 2 \cdot \mathsf{Adv}^{\mathsf{prf}}_{\mathsf{KDF},\mathcal{R}}(\lambda) \\ +\ \mathsf{Adv}^{\textit{prf}^{\textit{swap}}}_{\mathsf{KDF},\mathcal{R}}(\lambda) \end{array} \right), \\
n_P^2 \cdot n_S^2 \left(\begin{array}{l} \mathsf{CPAKEM}.\delta \\ +\ \mathsf{Adv}^{\textit{IND-CPA}}_{\mathsf{CPAKEM},\mathcal{R}}(\lambda) \\ +\ 4 \cdot \mathsf{Adv}^{\mathsf{prf}}_{\mathsf{KDF},\mathcal{R}}(\lambda) \\ +\ \mathsf{Adv}^{\textit{prf}^{\textit{swap}}}_{\mathsf{KDF},\mathcal{R}}(\lambda) \end{array} \right), \\
n_P^2 \cdot n_S^2 \left(\begin{array}{l} \mathsf{Adv}^{\textit{IND-CCA}}_{\mathsf{CCAKEM},\mathcal{R}}(\lambda) \\ +\ 7 \cdot \mathsf{Adv}^{\mathsf{prf}}_{\mathsf{KDF},\mathcal{R}}(\lambda) \\ +\ 2 \cdot \mathsf{Adv}^{\textit{prf}^{\textit{swap}}}_{\mathsf{KDF},\mathcal{R}}(\lambda) \end{array} \right), \\
n_P^2 \cdot n_S^2 \left(\begin{array}{l} \mathsf{Adv}^{\textit{IND-CCA}}_{\mathsf{CCAKEM},\mathcal{R}}(\lambda) \\ +\ 3 \cdot \mathsf{Adv}^{\mathsf{prf}}_{\mathsf{KDF},\mathcal{R}}(\lambda) \\ +\ 2 \cdot \mathsf{Adv}^{\textit{prf}^{\textit{swap}}}_{\mathsf{KDF},\mathcal{R}}(\lambda) \end{array} \right), \\
n_P^2 \cdot n_S^2 \left(\begin{array}{l} \frac{n_s}{2^\lambda} + \mathsf{Adv}^{\textit{IND-CCA}}_{\mathsf{CCAKEM},\mathcal{R}}(\lambda) \\ +\ 7 \cdot \mathsf{Adv}^{\mathsf{prf}}_{\mathsf{KDF},\mathcal{R}}(\lambda) \\ +\ \mathsf{Adv}^{\textit{prf}^{\textit{swap}}}_{\mathsf{KDF},\mathcal{R}}(\lambda) \end{array} \right)
\end{array} \right\}
\end{aligned}
$$

By combining some terms, we can simplify this equation to the following, simpler one:

$$
\begin{aligned}
&\mathsf{Adv}^{\textit{eCK-PFS-PSK}}_{\textit{pqWG},\mathsf{clean}_{\textit{eCK-PFS-PSK}},n_P,n_S,\mathcal{A}}(\lambda) \leq \\
&n_P^2 \cdot n_S \left(\begin{array}{l}
2 \cdot \mathsf{Adv}^{\textit{IND-CCA}}_{\mathsf{CCAKEM},\mathcal{R}}(\lambda) \; + \; 9 \cdot \mathsf{Adv}^{\mathsf{prf}}_{\mathsf{KDF},\mathcal{R}}(\lambda) \\
\quad + \; 4 \cdot \mathsf{Adv}^{\textit{prf}^{\textit{swap}}}_{\mathsf{KDF},\mathcal{R}}(\lambda) \; + \; 2 \cdot \mathsf{Adv}^{\textit{auth-aead}}_{\mathsf{AEAD},\mathcal{R}}(\lambda) \\
\qquad\qquad + \; 2 \cdot \frac{n_s}{2^\lambda} \\
\quad + \; n_S \cdot \max \left\{ \begin{array}{l}
\left(\begin{array}{l} \mathsf{CPAKEM}.\delta \\ \quad + \; \mathsf{Adv}^{\textit{IND-CPA}}_{\mathsf{CPAKEM},\mathcal{R}}(\lambda) \\ \quad + \; 4 \cdot \mathsf{Adv}^{\mathsf{prf}}_{\mathsf{KDF},\mathcal{R}}(\lambda) \\ \quad + \; \mathsf{Adv}^{\textit{prf}^{\textit{swap}}}_{\mathsf{KDF},\mathcal{R}}(\lambda) \end{array} \right), \\
\left(\begin{array}{l} \mathsf{Adv}^{\textit{IND-CCA}}_{\mathsf{CCAKEM},\mathcal{R}}(\lambda) \\ \quad + \; 7 \cdot \mathsf{Adv}^{\mathsf{prf}}_{\mathsf{KDF},\mathcal{R}}(\lambda) \\ \quad + \; 2 \cdot \mathsf{Adv}^{\textit{prf}^{\textit{swap}}}_{\mathsf{KDF},\mathcal{R}}(\lambda) \end{array} \right), \\
\left(\begin{array}{l} \mathsf{Adv}^{\textit{IND-CCA}}_{\mathsf{CCAKEM},\mathcal{R}}(\lambda) \\ \quad + \; 7 \cdot \mathsf{Adv}^{\mathsf{prf}}_{\mathsf{KDF},\mathcal{R}}(\lambda) \\ \quad + \; \mathsf{Adv}^{\textit{prf}^{\textit{swap}}}_{\mathsf{KDF},\mathcal{R}}(\lambda) \\ \quad + \; \frac{n_s}{2^\lambda} \end{array} \right)
\end{array} \right\}
\end{array} \right)
\end{aligned}
$$

At the cost of a (remarkably small) loss in tightness, we can further simplify this to the following:

$$
\begin{aligned}
&\mathsf{Adv}^{\textit{eCK-PFS-PSK}}_{\textit{pqWG},\mathsf{clean}_{\textit{eCK-PFS-PSK}},n_P,n_S,\mathcal{A}}(\lambda) \\
&\leq n_P^2 \cdot n_S \left(\begin{array}{rcl}
(7n_S + 9) & \cdot & \mathsf{Adv}^{\mathsf{prf}}_{\mathsf{KDF},\mathcal{R}}(\lambda) \\
+ \; (2n_S + 4) & \cdot & \mathsf{Adv}^{\textit{prf}^{\textit{swap}}}_{\mathsf{KDF},\mathcal{R}}(\lambda) \\
+ \; (n_S + 2) & \cdot & \mathsf{Adv}^{\textit{IND-CCA}}_{\mathsf{CCAKEM},\mathcal{R}}(\lambda) \\
+ \; n_S & \cdot & \mathsf{CPAKEM}.\delta \\
+ \; n_S & \cdot & \mathsf{Adv}^{\textit{IND-CPA}}_{\mathsf{CPAKEM},\mathcal{R}}(\lambda) \\
+ \; 2 & \cdot & \mathsf{Adv}^{\textit{auth-aead}}_{\mathsf{AEAD},\mathcal{R}}(\lambda) \\
+ \; (n_S + 2) & \cdot & \frac{n_s}{2^\lambda}
\end{array} \right)
\end{aligned}
$$

Note that for readability reasons, we drop the convention of including the cleanness predicate $\mathsf{clean}_{\mathsf{eCK\text{-}PFS\text{-}PSK}}$ in the advantage notation in what follows. We begin by dividing the proof into three separate cases (and denote with $\mathsf{Adv}^{\mathsf{eCK\text{-}PFS\text{-}PSK},C_l}_{\mathsf{pqWG},n_P,n_S,\mathcal{A}}(\lambda)$ the advantage of the adversary in winning the key-indistinguishability game in Case l) where the query $\mathsf{Test}(i,s)$ has been issued:

1. The session π_i^s (where $\pi_i^s.\rho = \texttt{init}$) has no contributive keyshare session.

2. The session π_i^s (where $\pi_i^s.\rho = \texttt{resp}$) has no contributive keyshare session.

3. The session π_i^s has a contributive keyshare session.

It follows then that $\mathsf{Adv}^{\mathsf{eCK\text{-}PFS\text{-}PSK}}_{\mathsf{pqWG},n_P,n_S,\mathcal{A}} \leq (\mathsf{Adv}^{\mathsf{eCK\text{-}PFS\text{-}PSK},C_1}_{\mathsf{pqWG},n_P,n_S,\mathcal{A}}(\lambda) + \mathsf{Adv}^{\mathsf{eCK\text{-}PFS\text{-}PSK},C_2}_{\mathsf{pqWG},n_P,n_S,\mathcal{A}}(\lambda) + \mathsf{Adv}^{\mathsf{eCK\text{-}PFS\text{-}PSK},C_3}_{\mathsf{pqWG},n_P,n_S,\mathcal{A}}(\lambda))$. We then bound the probability of each case, and show that under certain assumptions, the probability of the adversary winning in the key-indistinguishability game is negligible.

In the first two cases, we show that the adversary's probability in getting the session π_i^s to reach an "accept" state (and thus generate keys used in the real-or-random key indistinguishability game) is negligible, and since the adversary cannot cause the test session π_i^s to reach the accept state, the experiment will act identically regardless of whether the test bit b is 0 or 1, and thus the adversary's probability in winning the key indistinguishability game is negligible.

In the third case, we show that under certain assumptions, replacing the session keys with uniformly random, independent keys from the same distribution has a negligible chance of being detected and thus, the adversary's advantage in distinguishing the real-or-random key-indistinguishability game is also negligible. We begin with the first case.

4.7.1. Case 1: Test `init` session without contributive keyshare session

In this case we bound the probability that a test initiator session will accept when there exists no contributive keyshare session. Recall that a contributive keyshare session π_j^t exists for a session π_i^s when $\pi_j^t.kid$ is a substring of $\pi_i^s.m_r$. Informally, the test session π_i^s has not received keying material from an honest partner session, having either been modified or injected wholesale by the adversary.

Proof Sketch We begin first by adding an abort rule that triggers if there is ever a hash collision during the challenger's execution of any honest session. We follow by guessing the index of the test session, and adding an abort event that occurs if a Test query is directed to a session that does not have the index of the guessed session, and similarly, guess the party index of the intended

partner session. Afterwards, we add another abort event that occurs if the guessed test session π_i^s reaches the `reject` status. Since we already abort if the guessed session is not the session indicated by the Test query, and if the session π_i^s has reached the reject status, the $\mathsf{Test}(i, s)$ query will always respond with $\perp$, there is no difference in the adversary's advantage in the two games - any further queries that the adversary makes is responded to identically regardless of the sampling of the random test bit b.

We define an abort event $abort_{\texttt{accept}}$ that will occur if $\pi_i^s \leftarrow \texttt{accept}$. The following games then are designed to bound the probability of $abort_{\texttt{accept}}$ occurring to be negligibly close to zero. Note that from this game onwards, the adversary is unable to make a $\mathsf{CorruptASK}(j)$ query, since we now abort the game when the session π_i^s reaches a status that is not `active`, and by the Case 1 definition (a test session without a contributive keyshare session) and the cleanness predicate $\mathsf{clean}_{\mathsf{eCK\text{-}PFS\text{-}PSK}}$, the adversary can only win by not issuing a $\mathsf{CorruptASK}(j)$ query before the test session completes. We can now (cleverly) embed CCAKEM challenge values from the IND-CCA challenger into the long-term asymmetric keys of the party P_j without needing to address the adversary's ability to issue a $\mathsf{CorruptASK}(j)$ query.

We then replace the values C_3, κ_3 with uniformly random and independent values $\widetilde{C_3}, \widetilde{\kappa_3}$, and argue that any adversary capable of distinguishing this change would be able to break either the prf or the IND-CCA assumption. In the next game we replace the values C_4, κ_4 with uniformly random and independent values $\widetilde{C_4}, \widetilde{\kappa_4}$, and argue that any adversary capable of distinguishing this change would be able to break the prf security of KDF.

In a similar fashion, we use a chain of prf challengers to replace C_6, C_7, C_8 and finally C_9, tmp, κ_9 with uniformly random and independent values $\widetilde{C_6}$, $\widetilde{C_7}$, $\widetilde{C_8}$ and $\widetilde{C_9}$, $\widetilde{tmp}$, $\widetilde{\kappa_9}$. We argue that any adversary $\mathcal{A}$ capable of distinguishing these changes can be turned into a successful distinguishing adversary against the prf security of KDF.

In the final game hop, we use the fact that $\widetilde{\kappa_9}$ is a uniformly random and independent value to embed $\widetilde{\kappa_9}$ within an aead challenger, and add an abort rule $abort_{dec}$ that triggers when the test session π_i^s decrypts a `zero` ciphertext received in the `RespHello` message. To do so, we use the aead decryption oracle to replace concrete decryptions performed in the test session. Logically then, since the $\widetilde{\kappa_9}$ value is internal to the aead challenger, if `zero` decrypts correctly, then $\mathcal{A}$ has managed to produce a ciphertext $\mathsf{AEAD.Enc}(\widetilde{\kappa_9}, 0, H_9, \emptyset)$ that has not been the result of an encryption oracle query on $(\emptyset, H_9)$, and we can use `zero`, to break the aead security of the AEAD scheme. We note that since $\widetilde{\kappa_9}$ is already a uniformly random and independent value, that this

change is sound, and that the probability of $abort_{dec}$ triggering is bound by the probability of the adversary breaking the aead security of AEAD.

Since a session with role $\pi_i^s.\rho = \texttt{init}$ will only accept if it receives a ciphertext zero that decrypts correctly, and $abort_{dec}$ triggers if such a ciphertext decrypts correctly, then the probability of π_i^s reaching an accept state is 0 in the final game, and the adversary cannot force a session π_i^s to accept without an honest partner π_j^t. We show this using the following sequence of games:

Game 0 This is a standard eCK-PFS-PSK game. Thus we have

$$\mathsf{Adv}^{\mathsf{eCK\text{-}PFS\text{-}PSK},C_1}_{\mathsf{pqWG},n_P,n_S,\mathcal{A}}(\lambda) = \Pr\left[\text{break}_0\right].$$

Game 1 In this game, we guess the index (i, s) of the session π_i^s, and abort if during the execution of the experiment, a query $\mathsf{Test}(i^*, s^*)$ is received and $(i^*, s^*) \neq (i, s)$. Thus:

$$\Pr\left[\text{break}_0\right] \leq n_P \cdot n_S \cdot \Pr\left[\text{break}_1\right]$$

Game 2 In this game, we guess the party of the intended partner of the test session π_i^s, and abort if $\pi_i^s.pid \neq j$. Thus

$$\Pr\left[\text{break}_1\right] \leq n_P \cdot \Pr\left[\text{break}_2\right].$$

Game 3 In this game, we abort if the session π_i^s sets the status $\pi_i^s.\alpha \leftarrow \texttt{reject}$. Note that by *Game 2* we abort if the Test query is ever issued to a session that is not π_i^s. If the session π_i^s ever reaches the status $\pi_i^s.\alpha \leftarrow \texttt{reject}$, then the $\mathsf{Test}(i, s)$ query will be rejected by the challenger as specified in Figure A.1. Note that the difference between the adversary's advantage in *Game 2* and *Game 3* is 0: The sampling of the test bit b by the challenger only affects the response to the $\mathsf{Test}(i, s)$ query, which is always rejected if $\pi_i^s.\alpha = \texttt{reject}$. Thus

$$\Pr\left[\text{break}_2\right] = \Pr\left[\text{break}_3\right].$$

Game 4 In this game we define an abort event $abort_{\texttt{accept}}$ that triggers if the status of the test session $\pi_i^s \leftarrow \texttt{accept}$. It is clear then that

$$\Pr\left[\text{break}_3\right] = \Pr(abort_{\texttt{accept}}).$$

In the following sequence of games, we show that the probability of the abort event triggering (i.e. $\Pr(abort_{\texttt{accept}})$) is negligibly close to zero.

Game 5 In this game we replace the computation of C_3, κ_3 with uniformly random and independent values $\widetilde{C_3}, \widetilde{\kappa_3}$. We note that the replacement of the

sym-ms-PRF-ODH assumptions with the more standard IND-CCA assumption for KEMs forces us to split the original hybrid into three. This is necessary because of the more convoluted combination of the static keys with both the other parties static and ephemeral keys and because the application of the KDF to the shared-secret is not part of the IND-CCA game while it was part of the PRF-ODH game. As such we first replace the pseudo-random value used for key-encapsulation with CCAKEM with a truly random value (*Game 5a*) and then replace `ct1` with a random value $\mathtt{k}^*$ (*Game 5b*). After that we replace the output of the KDF that this value is passed to with a random one (*Game 5c*). The reason for why we split the hybrid instead of inserting new ones is that we want to stay consistent with the numbering of the hybrids in the original proof.

The one case where we will deviate from the original numbering-scheme is in the labels for the "break"-events in **Case 1**: The original proof numbers theses such that $\Pr[\text{break}_4]$ is the probability that the fifth hybrid is broken; in all other cases the numbers coincide however. Because we believe that skipping break_4 and increasing all following indices by one is more readable and since this is what we do in the full version, the indices in our proof don't match the ones from the original proof by Dowling and Paterson. (Again: This does not affect cases 2 and 3.)

In ***Game 5a*** we replace the value $\hat{r} := \mathsf{KDF}(\sigma_i, r_i)$ passed to CCAKEM.Enc for the computation of `ct1` and `shk1` with a random bitstring $\hat{r}'$.

By the definition of this case, we know that at least one of r_i and σ_i is random and uncorrupted.

In the first case (r_i is unknown to the adversary), we initialize a $\mathsf{prf}^{\mathsf{swap}}$ challenger, query σ_i, and use the output $\tilde{r}$ from the $\mathsf{prf}^{\mathsf{swap}}$ challenger to replace the computation of $\hat{r}$. By the definition of this case r_i is a uniformly random and independent value, therefore this replacement is sound. If the test bit sampled by the $\mathsf{prf}^{\mathsf{swap}}$ challenger is 0, then $\hat{r} \leftarrow \mathsf{KDF}(\sigma_i, r_i)$ and we are in *Game 4*. If the test bit sampled by the $\mathsf{prf}^{\mathsf{swap}}$ challenger is 1, then $\hat{r} \xleftarrow{\$} \{0,1\}^{|\mathsf{KDF}|}$ is a truly random value and we are in *Game 5a*.

For the second case we first establish that r_i, while being (potentially) known to the adversary is still fresh in the sense that $\mathsf{KDF}(\sigma_i, r_i)$ has never been evaluated: Since r_i is a random value, there is a chance that it could be sampled in another session. This probability can be upper-bounded by the total number of sessions divided by the number of possible values, namely $\frac{n_S}{2^\lambda}$ (which when multiplied by the number of sessions results in the famous approximation of the birthday-bound $\frac{n_S^2}{2^\lambda}$).

Given that, we initialize a prf challenger and replace all computations of $\mathsf{KDF}(\sigma_i, \cdot)$ with queries to the challenger. By the definition of this case σ_i is a uniformly random and independent value, therefore this replacement is sound. If the test bit sampled by the prf challenger is 0, then $\hat{r} \leftarrow \mathsf{KDF}(\sigma_i, r_i)$ and we are in *Game 4*. If the test bit sampled by the prf challenger is 1, then $\hat{r} \xleftarrow{\$} \{0,1\}^{|\mathsf{KDF}|}$ is a truly random value. Since we established furthermore that r_i is not used with σ_i in any other session, $\hat{r}$ is furthermore independent of all other $\hat{r}$ in other sessions, therefore we are in *Game 5a*.

Thus any adversary capable of distinguishing this change can be turned into a successful adversary against the prf security or the $\mathsf{prf}^{\mathsf{swap}}$ security of KDF, and we find:

$$\begin{aligned}&\Pr(abort_{\mathtt{accept}})\\ &\leq \frac{n_S}{2^\lambda} + \mathsf{Adv}^{\mathsf{prf}}_{\mathsf{KDF},\mathcal{R}}(\lambda) + \mathsf{Adv}^{\mathsf{prf}^{\mathsf{swap}}}_{\mathsf{KDF},\mathcal{R}}(\lambda) + \Pr\left[\mathrm{break}_{5a}\right]\end{aligned}$$

In ***Game 5b*** we replace the computation of `shk1` by sampling the value uniformly at random from the space of shared secrets of the KEM and ignoring the second output of $\mathsf{CCAKEM.Enc}(\mathtt{spk}_r)$. To show that this is undetectable under the IND-CCA-assumption of the used KEM, we interact with an IND-CCA challenger in the following way: Note that by *Game 1*, we know at the beginning of the experiment the index of session π_i^s such that $\mathsf{Test}(i,s)$ is issued by the adversary. Similarly, by *Game 2*, we know at the beginning of the experiment the index of the intended partner P_j of the session π_i^s. Thus, we initialize an IND-CCA challenger and use the received public-key $\mathtt{pk}^*$ as long-term public-key of party P_j and give it with all other (honestly generated) public keys to the adversary. Note that by *Game 4* and the definition of this case, $\mathcal{A}$ is not able to issue a $\mathsf{CorruptASK}(j)$ query, as we abort if $\pi_i^s.\alpha \leftarrow \mathtt{reject}$ and abort if $\pi_i^s.\alpha \leftarrow \mathtt{accept}$. Thus we will not need to reveal the private key $\mathtt{sk}^*$ of the challenge public-key to $\mathcal{A}$. However we must account for all sessions t such that π_j^t must use the private key for computations. In our version of WireGuard, the long-term private keys are used to compute the following:

- In sessions where P_j acts as the initiator:
  $C_8 \leftarrow \mathsf{KDF}(C_6, \mathsf{CCAKEM.Dec}(\mathtt{ssk}_i, \mathtt{ct3}))$
- In sessions where P_j acts as the responder:
  $C_3, \kappa_3 \leftarrow \mathsf{KDF}(C_2, \mathsf{CCAKEM.Dec}(\mathtt{ssk}_r, \mathtt{ct1}))$

(Note that these are fewer cases than in the original proof because we don't combine static and ephemeral keys directly.) Dealing with the challenger's computation of these values will be done in two ways:

- The encapsulation was created by another honest party. The challenger can then use its own internal knowledge of the encapsulated value to complete the computations.
- The encapsulation was not created by another honest party, but by the adversary and the challenger is therefore unaware of the encapsulated value.

In the second case, the challenger can instead use the decapsulation-oracle provided by the CCA-challenger, specifically querying $\mathsf{CCAKEM.Dec}(\mathtt{ctX})$, (where $\mathtt{ctX}$ is the relevant encapsulation) which will output $\mathtt{shkX}$ using the CCA challenger's internal knowledge of $\mathtt{sk}^*$.

During session i we request a challenge consisting of a ciphertext and a candidate shared secret $(\mathtt{c}^*, \mathtt{k}^*)$ from the IND-CCA challenger and use those values in place of $\mathtt{ct1}$ and $\mathtt{shk1}$. Given the definition of the IND-CCA game, there are two cases:

- If the test bit sampled by the IND-CCA challenger is 0, then $\mathtt{k}^*$ is indeed the shared secret encapsulated in $\mathtt{c}^*$ and we are in *Game 5a*.
- If the test bit sampled by the IND-CCA challenger is 1, then $\mathtt{k}^*$ is not the shared secret encapsulated in $\mathtt{c}^*$ but sampled uniformly at random from the space of shared secrets and we are in *Game 5b*.

Thus, any adversary capable of distinguishing this change can be turned into a successful adversary against the IND-CCA security of the used KEM and we find:

$$\Pr\left[\text{break}_{5a}\right] \leq \mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathsf{CCAKEM},\mathcal{R}}(\lambda) + \Pr\left[\text{break}_{5b}\right]$$

In ***Game 5c*** we replace the values of C_3, κ_3 with uniformly random and independent values $\widetilde{C_3}, \widetilde{\kappa_3} \xleftarrow{\$} \{0,1\}^{|\mathsf{KDF}|}$ (where $\{0,1\}^{|\mathsf{KDF}|}$ is the output space of the KDF) used in the protocol execution of the test session. Specifically, we initialize a $\mathsf{prf}^{\mathsf{swap}}$ challenger and query $\mathtt{shk1}$, and use the output $\widetilde{C_3}, \widetilde{\kappa_3}$ from the $\mathsf{prf}^{\mathsf{swap}}$ challenger to replace the computation of C_3, κ_3. Since by *Game 5b*, $\mathtt{shk1}$ is a uniformly random and independent value, this replacement is sound. If the test bit sampled by the $\mathsf{prf}^{\mathsf{swap}}$ challenger is 0, then $\widetilde{C_3}, \widetilde{\kappa_3} \leftarrow$

$\mathsf{KDF}(C_2, \texttt{shk1})$ and we are in *Game 5b*. If the test bit sampled by the $\mathsf{prf}^{\mathsf{swap}}$ challenger is 1, then $\widetilde{C_3}, \widetilde{\kappa_3} \xleftarrow{\$} \{0,1\}^{|\mathsf{KDF}|}$ and we are in *Game 5c*.

Thus any adversary capable of distinguishing this change can be turned into a successful adversary against the $\mathsf{prf}^{\mathsf{swap}}$ security of KDF, and we find:

$$\Pr[\text{break}_{5b}] \leq \mathsf{Adv}^{\mathsf{prf}^{\mathsf{swap}}}_{\mathsf{KDF},\mathcal{R}}(\lambda) + \Pr[\text{break}_{5c}]$$

Game 6 In this game we replace the values C_4, κ_4 with uniformly random and independent values $\widetilde{C_4}, \widetilde{\kappa_4} \xleftarrow{\$} \{0,1\}^{\mathsf{KDF}}$ (where $\{0,1\}^{\mathsf{KDF}}$ is the output space of the KDF) used in the protocol execution of the test session. Specifically, we initialize a prf challenger and query psk, and use the output $\widetilde{C_4}, \widetilde{\kappa_4}$ from the prf challenger to replace the computation of C_4, κ_4. Since by *Game 5c*, $\widetilde{C_3}$ is a uniformly random and independent value, this replacement is sound. If the test bit sampled by the prf challenger is 0, then $\widetilde{C_4}, \widetilde{\kappa_4} \leftarrow \mathsf{KDF}(C_3, psk)$ and we are in *Game 5c*. If the test bit sampled by the prf challenger is 1, then $\widetilde{C_4}, \widetilde{\kappa_4} \xleftarrow{\$} \{0,1\}^{|\mathsf{KDF}|}$ and we are in *Game 6*. Thus any adversary $\mathcal{A}$ capable of distinguishing this change can be turned into a successful adversary against the prf security of KDF, and we find:

$$\Pr[\text{break}_{5c}] \leq \mathsf{Adv}^{\mathsf{prf}}_{\mathsf{KDF},\mathcal{R}}(\lambda) + \Pr[\text{break}_{6}]$$

Game 7 In this game we replace the value C_6 with a uniformly random and independent value $\widetilde{C_6} \xleftarrow{\$} \{0,1\}^{|\mathsf{KDF}|}$ (where $\{0,1\}^{|\mathsf{KDF}|}$ is the output space of KDF) used in the protocol execution of the test session. Specifically, we initialize a prf challenger, query it with $\texttt{ct2}$, and use the output $\widetilde{C_6}$ from the prf challenger to replace the computation of C_6. Since by *Game 6*, $\widetilde{C_4}$ is a uniformly random and independent value, this replacement is sound. If the test bit sampled by the prf challenger is 0, then $\widetilde{C_6} \leftarrow \mathsf{prf}(C_4, \texttt{ct2})$ and we are in *Game 6*. If the test bit sampled by the prf challenger is 1, then $\widetilde{C_6} \xleftarrow{\$} \{0,1\}^{|\mathsf{KDF}|}$ and we are in *Game 7*. Thus any adversary $\mathcal{A}$ capable of distinguishing this change can be turned into a successful adversary against the prf security of KDF, and we find:

$$\Pr[\text{break}_{6}] \leq \mathsf{Adv}^{\mathsf{prf}}_{\mathsf{KDF},\mathcal{R}}(\lambda) + \Pr[\text{break}_{7}]$$

Game 8 As in previous games, we replace the computation of C_7 with a uniformly random value $\widetilde{C_7}$ from the same distribution, in the challenger's execution of the test session π^s_i. We do so by interacting with a prf challenger

in the following way: When it is time to compute $C_7 \leftarrow \mathsf{KDF}(\widetilde{C_6}, \mathtt{shk2})$ we instead initialize a prf challenger and query it with shk2. We note that by *Game 7*, $\widetilde{C_6}$ is a uniformly random and independent value, and thus this replacement is sound.If the random bit b sampled by the prf challenger is 0, then $\widetilde{C_7} \leftarrow \mathsf{KDF}(\widetilde{C_6}, \mathtt{shk2})$ and we are in *Game 7*. If the random bit b sampled by the prf challenger is 1, then $\widetilde{C_7} \xleftarrow{\$} \{0,1\}^{|\mathsf{KDF}|}$ and we are in *Game 8* . Any adversary $\mathcal{A}$ capable of distinguishing this change in the experiment can be turned into an algorithm against the prf security of KDF, and thus

$$\Pr\left[\mathrm{break}_7\right] \leq \mathsf{Adv}^{\mathsf{prf}}_{\mathsf{KDF},\mathcal{R}}(\lambda) + \Pr\left[\mathrm{break}_8\right].$$

Game 9 As in previous games, we replace the computation of C_8 with a uniformly random value $\widetilde{C_8}$ from the same distribution, in the challenger's execution of the test session π_i^s. We do so by interacting with a prf challenger in the following way: When it is time to compute $C_8 \leftarrow \mathsf{KDF}(\widetilde{C_7}, \mathtt{shk3})$ we instead initialize a prf challenger and query it with shk3. We note that by *Game 8*, $\widetilde{C_7}$ is a uniformly random and independent value, and thus this replacement is sound.If the random bit b sampled by the prf challenger is 0, then $C_8 \leftarrow \mathsf{KDF}(\widetilde{C_7}, \mathtt{shk3})$ and we are in *Game 8*. If the random bit b sampled by the prf challenger is 1, then $\widetilde{C_8} \xleftarrow{\$} \{0,1\}^{|\mathsf{KDF}|}$ and we are in *Game 9*. Any adversary $\mathcal{A}$ capable of distinguishing this change in the experiment can be turned into an algorithm against the prf security of KDF, and thus

$$\Pr\left[\mathrm{break}_8\right] \leq \mathsf{Adv}^{\mathsf{prf}}_{\mathsf{KDF},\mathcal{R}}(\lambda) + \Pr\left[\mathrm{break}_9\right].$$

Game 10 As in previous games, we replace the computation of C_9, tmp, κ_9 with uniformly random values $\widetilde{C_9}, \widetilde{tmp}, \widetilde{\kappa_9}$ from the same distribution, in the challenger's execution of the test session π_i^s and its partner session π_j^t. We do so by interacting with a KDF challenger in the following way: When it is time to compute $C_9, tmp, \kappa_9 \leftarrow \mathsf{KDF}(\widetilde{C_8}, psk)$ we instead initialize a prf challenger and query it with psk. We note that by *Game 9*, $\widetilde{C_8}$ is a uniformly random and independent value, and thus this replacement is sound.If the random bit b sampled by the prf challenger is 0, then we are in *Game 9*. If the random bit b sampled by the prf challenger is 1, then we are in *Game 10*. Any adversary $\mathcal{A}$ capable of distinguishing this change in the experiment can be turned into an algorithm against the prf security of KDF, and thus

$$\Pr\left[\mathrm{break}_9\right] \leq \mathsf{Adv}^{\mathsf{prf}}_{\mathsf{KDF},\mathcal{R}}(\lambda) + \Pr\left[\mathrm{break}_{10}\right].$$

Game 11 In this game, the test session π_i^s will only set $\pi_i^s.\alpha \leftarrow \texttt{accept}$ if the adversary is able to produce a value $\texttt{zero} = \mathsf{AEAD}(\widetilde{\kappa_9}, 0, H_9, \emptyset)$ that decrypts correctly. In this game, we now initialize an aead challenger to decrypt `RespHello.zero` ciphertexts in the test session π_i^s. By *Game 10* that $\widetilde{\kappa_9}$ is a uniformly random and independent value, and thus this change is undetectable. Since the $\widetilde{\kappa_9}$ is internal to the aead challenger, then it follows that the adversary capable of forging such a `zero` ciphertext breaks the security of the AEAD scheme. We find that

$$\Pr\left[\text{break}_{10}\right] = \mathsf{Adv}^{\text{auth-aead}}_{\mathsf{AEAD},\mathcal{R}}(\lambda).$$

Thus

$$\begin{aligned}\Pr(\mathit{abort}_{\texttt{accept}}) \leq (&\frac{n_S}{2^\lambda} + \mathsf{Adv}^{\text{IND-CCA}}_{\mathsf{CCAKEM},\mathcal{R}}(\lambda) \\ &+ 6 \cdot \mathsf{Adv}^{\text{prf}}_{\mathsf{KDF},\mathcal{R}}(\lambda) + 2 \cdot \mathsf{Adv}^{\text{prf}^{\text{swap}}}_{\mathsf{KDF},\mathcal{R}}(\lambda) \\ &+ \mathsf{Adv}^{\text{auth-aead}}_{\mathsf{AEAD},\mathcal{R}}(\lambda))\end{aligned}$$

It follows then

$$\begin{aligned}\mathsf{Adv}^{\text{eCK-PFS-PSK},C_1}_{\mathsf{pqWG},n_P,n_S,\mathcal{A}}(\lambda) \leq\ & n_P^2 \cdot n_S \Big(\frac{n_S}{2^\lambda} \\ &+ \mathsf{Adv}^{\text{IND-CCA}}_{\mathsf{CCAKEM},\mathcal{R}}(\lambda) \\ &+ 6 \cdot \mathsf{Adv}^{\text{prf}}_{\mathsf{KDF},\mathcal{R}}(\lambda) \\ &+ 2 \cdot \mathsf{Adv}^{\text{prf}^{\text{swap}}}_{\mathsf{KDF},\mathcal{R}}(\lambda) \\ &+ \mathsf{Adv}^{\text{aead}}_{\mathsf{AEAD},\mathcal{R}}(\lambda) \Big).\end{aligned}$$

4.7.2. Case 2: Test `resp` session without contributive keyshare partner

In this case we bound the probability that a session π_i^s such that $\pi_i^s.\rho = \texttt{resp}$ will accept when there exists no contributive keyshare partner. Recall that an contributive keyshare partner exists for a session π_i^s when for some session π_j^t, $\pi_j^t.\mathit{kid}$ is a substring of $\pi_i^s.m_r$. Informally, the test session π_i^s has not received the keyshares that were honestly generated by another session, having either been modified or injected wholesale by the adversary.

Proof sketch We begin by guessing the index of the test session, and adding an abort event that occurs if a Test query is directed to a session

that does not have the index of the guessed session, and similarly, guess the party index of the intended partner session. Afterwards, we add another abort event that occurs if the guessed test session π_i^s reaches the `reject` status. Since we already abort if the guessed session is not the session indicated by the Test query, and if the session π_i^s has reached the reject status, the $\mathsf{Test}(i, s)$ query will always respond with $\perp$, there is no difference in the adversary's advantage in the two games - any further queries that the adversary makes is responded to identically regardless of the sampling of the random test bit b.

We define an abort event $abort_{\texttt{accept}}$ that will occur if $\pi_i^s \leftarrow \texttt{accept}$. The following games then are designed to bound the probability of $abort_{\texttt{accept}}$ occurring to be negligibly close to zero. Note that from this game onwards, the adversary is unable to make a $\mathsf{CorruptASK}(j)$ query, since we now abort the game when the session π_i^s reaches a status that is not `active`, and by the Case 1 definition (a test session without a contributive keyshare session) and the cleanness predicate $\mathsf{clean}_{\mathsf{eCK\text{-}PFS\text{-}PSK}}$, the adversary can only win by not issuing a $\mathsf{CorruptASK}(j)$ query before the test session completes. We can now (cleverly) embed CCAKEM challenge values from the IND-CCA challenger into the long-term asymmetric keys of the party P_j without needing to address the adversary's ability to issue a $\mathsf{CorruptASK}(j)$ query.

We then replace the value C_8 with a uniformly random and independent value $\widetilde{C_8}$, and argue that any adversary capable of distinguishing this change would be able to break either the prf or the IND-CCA assumption. In the next game we replace the values C_9, tmp, κ_4 with uniformly random and independent values $\widetilde{C_9}, \widetilde{tmp}, \widetilde{\kappa_4}$, and argue that any adversary capable of distinguishing this change would be able to break the PRF assumption. In a similar fashion, we replace the values C_{10}, κ_{10} with uniformly random and independent values $\widetilde{C_{10}}, \widetilde{\kappa_{10}}$ and again argue that any distinguishing adversary can be turned into an adversary against the PRF assumption. Finally, we argue that the test session π_i^s will only reach an `accept` state (and trigger the $abort_{\texttt{accept}}$ event) if it receives a value $\texttt{conf} = \mathsf{AEAD.Enc}(\widetilde{\kappa_{10}}, 0, H_{10}, \emptyset)$. We use the fact that $\widetilde{\kappa_{10}}$ is a uniformly random and independent value to embed $\widetilde{\kappa_{10}}$ within an `aead-au` challenger, and add an abort rule $abort_{dec}$ that triggers if the `conf` ciphertext received in the `SenderConf` message would decrypt without error. Logically then, since the $\widetilde{\kappa_{10}}$ value is internal to the `aead` challenger, if `conf` would decrypt correctly, then $\mathcal{A}$ has managed to produce a ciphertext $\mathsf{AEAD.Enc}(\widetilde{\kappa_{10}}, 0, H_{10}, \emptyset)$ that has not been the result of an encryption oracle query on $(0, \emptyset, H_{10})$, and we can use `zero` to break the `aead-au` security of the AEAD scheme. We note that since $\widetilde{\kappa_{10}}$ is already a uniformly random

and independent value, that this change is sound, and that the probability of $abort_{dec}$ triggering is bound by the probability of adversary breaking the aead-au security of AEAD.

Since a session with role $\pi_i^s.\rho = \texttt{resp}$ will only accept if it receives a ciphertext conf that decrypts correctly, and $abort_{dec}$ triggers if such a ciphertext decrypts correctly, then the probability of π_i^s reaching an accept state is 0 in the final game, and the adversary cannot force a session π_i^s to accept without a contributive keyshare partner π_j^t.

Game 0 This is a standard eCK-PFS-PSK game. Thus we have:

$$\mathsf{Adv}^{\mathsf{eCK\text{-}PFS\text{-}PSK},C_2}_{\mathsf{pqWG},n_P,n_S,\mathcal{A}}(\lambda) = \Pr\left[\text{break}_0\right]$$

Game 1 In this game, we guess the index (i, s) of the session π_i^s, and abort if during the execution of the experiment, a query $\mathsf{Test}(i^*, s^*)$ is received and $(i^*, s^*) \neq (i, s)$. Thus:

$$\Pr\left[\text{break}_0\right] \leq n_P \cdot n_S \cdot \Pr\left[\text{break}_1\right]$$

Game 2 In this game, we guess the party of the intended partner of the test session π_i^s, and abort if $\pi_i^s.pid \neq j$. Thus:

$$\Pr\left[\text{break}_1\right] \leq n_P \cdot \Pr\left[\text{break}_2\right]$$

Game 3 In this game, we abort if the session π_i^s sets the status $\pi_i^s.\alpha \leftarrow \texttt{reject}$. Note that by *Game 1* we abort if the Test query is ever issued to a session that is not π_i^s. If the session π_i^s ever reaches the status $\pi_i^s.\alpha \leftarrow \texttt{reject}$, then the $\mathsf{Test}(i, s)$ query will be rejected by the challenger as specified in Figure A.1. Note that the difference between the adversary's advantage in *Game 2* and *Game 3* is 0 as the sampling of the test bit b by the challenger only affects the response to the $\mathsf{Test}(i, s)$ query, which is always rejected if $\pi_i^s.\alpha = \texttt{reject}$. Thus:

$$\Pr\left[\text{break}_2\right] = \Pr\left[\text{break}_3\right]$$

Game 4 In this game we define an abort event $abort_{\texttt{accept}}$ that triggers if the status of the test session $\pi_i^s \leftarrow \texttt{accept}$. It is clear then that

$$\Pr\left[\text{break}_3\right] \leq \Pr(abort_{\texttt{accept}}) + \Pr\left[\text{break}_4\right]$$

and additionally that $\Pr\left[\text{break}_4\right] = 0$, since all responses to the adversary are identical regardless of the sampling of the test bit b. In the following

sequence of games, we show that the probability of the abort event triggering (i.e. $\Pr(abort_{\mathtt{accept}})$) is negligibly close to zero.

Game 5 In this game we replace the computation of C_8 with uniformly random and independent values $\widetilde{C_8}$. This works very similar to *Game 5* of **Case 1** and mostly changes labels. For the same reason as back then, we also split this game into three subhybrids numbered 5a, 5b and 5c.

In ***Game 5a*** we replace the value $\hat{r} := \mathsf{KDF}(\sigma_r, r_r)$ passed to CCAKEM.Enc for the computation of `ct1` and `shk1` with a random bitstring $\hat{r}'$.

By the definition of this case, we know that at least one of r_r and σ_r is random and uncorrupted.

In the first case (r_r is unknown to the adversary), we initialize a $\mathsf{prf}^{\mathsf{swap}}$ challenger, query σ_r, and use the output $\tilde{r}$ from the $\mathsf{prf}^{\mathsf{swap}}$ challenger to replace the computation of $\hat{r}$. By the definition of this case r_r is a uniformly random and independent value, therefore this replacement is sound. If the test bit sampled by the $\mathsf{prf}^{\mathsf{swap}}$ challenger is 0, then $\hat{r} \leftarrow \mathsf{KDF}(\sigma_r, r_r)$ and we are in *Game 4*. If the test bit sampled by the $\mathsf{prf}^{\mathsf{swap}}$ challenger is 1, then $\hat{r} \xleftarrow{\$} \{0,1\}^{|\mathsf{KDF}|}$ is a truly random value and we are in *Game 5a*.

For the second case we first establish that r_r, while being (potentially) known to the adversary is still fresh in the sense that $\mathsf{KDF}(\sigma_r, r_r)$ has never been evaluated: Since r_r is a random value, there is a chance that it could be sampled in another session. This probability can be upper-bounded by the total number of sessions divided by the number of possible values, namely $\frac{n_S}{2^\lambda}$ (which when multiplied by the number of sessions results in the famous approximation of the birthday-bound $\frac{n_S^2}{2^\lambda}$).

Given that, we initialize a prf challenger and replace all computations of $\mathsf{KDF}(\sigma_r, \cdot)$ with queries to the challenger. By the definition of this case σ_r is a uniformly random and independent value, therefore this replacement is sound. If the test bit sampled by the prf challenger is 0, then $\hat{r} \leftarrow \mathsf{KDF}(\sigma_r, r_r)$ and we are in *Game 4*. If the test bit sampled by the prf challenger is 1, then $\hat{r} \xleftarrow{\$} \{0,1\}^{|\mathsf{KDF}|}$ is a truly random value. Since we established furthermore that r_r is not used with σ_r in any other session, $\hat{r}$ is furthermore independent of all other $\hat{r}$ in other sessions, therefore we are in *Game 5a*.

Thus any adversary capable of distinguishing this change can be turned into a successful adversary against the prf security or the $\mathsf{prf}^{\mathsf{swap}}$ security of KDF, and we find:

$$\begin{aligned}&\Pr(abort_{\mathtt{accept}})\\ &\leq \frac{n_S}{2^\lambda} + \mathsf{Adv}^{\mathsf{prf}}_{\mathsf{KDF},\mathcal{R}}(\lambda) + \mathsf{Adv}^{\mathsf{prf}^{\mathsf{swap}}}_{\mathsf{KDF},\mathcal{R}}(\lambda) + \Pr\left[\mathrm{break}_{5a}\right]\end{aligned}$$

In ***Game 5b*** we replace the computation of $\texttt{shk3}$ by sampling the value uniformly at random from the space of shared secrets of the KEM and ignoring the second output of $\mathsf{CCAKEM.Enc}(\texttt{spk}_r)$. To show that this is undetectable under the IND-CCA-assumption of the used KEM, we interact with an IND-CCA challenger in the following way: Note that by *Game 2*, we know at the beginning of the experiment the index of session π_i^s such that $\mathsf{Test}(i, s)$ is issued by the adversary. Similarly, by *Game 3*, we know at the beginning of the experiment the index of the intended partner P_j of the session π_i^s. Thus, we initialize an IND-CCA challenger and use the received public-key $\texttt{pk}^*$ as long-term public-key of party P_j and give it with all other (honestly generated) public keys to the adversary. Note that by *Game 4* and the definition of this case, $\mathcal{A}$ is not able to issue a $\mathsf{CorruptASK}(j)$ query, as we abort if $\pi_i^s.\alpha \leftarrow \texttt{reject}$ and abort if $\pi_i^s.\alpha \leftarrow \texttt{accept}$. Thus we will not need to reveal the private key $\texttt{sk}^*$ of the challenge public-key to $\mathcal{A}$. However we must account for all sessions t such that π_j^t must use the private key for computations. In our version of WireGuard, the long-term private keys are used to compute the following:

- In sessions where P_j acts as the initiator:
  $C_8 \leftarrow \mathsf{KDF}(C_6, \mathsf{CCAKEM.Dec}(\texttt{ssk}_i, \texttt{ct3}))$
- In sessions where P_j acts as the responder:
  $C_3, \kappa_3 \leftarrow \mathsf{KDF}(C_2, \mathsf{CCAKEM.Dec}(\texttt{ssk}_r, \texttt{ct1}))$

(Note that these are fewer cases than in the original proof because we don't combine static and ephemeral keys directly.) Dealing with the challenger's computation of these values will be done in two ways:

- The encapsulation was created by another honest party. The challenger can then use its own internal knowledge of the encapsulated value to complete the computations.
- The encapsulation was not created by another honest party, but by the adversary and the challenger is therefore unaware of the encapsulated value.

In the second case, the challenger can instead use the decapsulation-oracle provided by the CCA-challenger, specifically querying $\mathsf{CCAKEM.Dec}(\texttt{ctX})$, (where $\texttt{ctX}$ is the relevant encapsulation) which will output $\texttt{shkX}$ using the CCA challenger's internal knowledge of $\texttt{sk}^*$.

During session i we request a challenge consisting of a ciphertext and a candidate shared secret $(\texttt{c}^*, \texttt{k}^*)$ from the IND-CCA challenger and use those

values in place of $\texttt{ct3}$ and $\texttt{shk3}$. Given the definition of the IND-CCA game, there are two cases:

- If the test bit sampled by the IND-CCA challenger is 0, then $\texttt{k}^*$ is indeed the shared secret encapsulated in $\texttt{c}^*$ and we are in *Game 5a*.
- If the test bit sampled by the IND-CCA challenger is 1, then $\texttt{k}^*$ is not the shared secret encapsulated in $\texttt{c}^*$ but sampled uniformly at random from the space of shared secrets and we are in *Game 5b*.

Thus, any adversary capable of distinguishing this change can be turned into a successful adversary against the IND-CCA security of the used KEM and we find:

$$\Pr[\text{break}_{5a}] \leq \mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathsf{CCAKEM},\mathcal{R}}(\lambda) + \Pr[\text{break}_{5b}]$$

In ***Game 5c*** we replace the values of C_8 with uniformly random and independent values $\widetilde{C_8} \xleftarrow{\$} \{0,1\}^{|\mathsf{KDF}|}$ (where $\{0,1\}^{|\mathsf{KDF}|}$ is the output space of the KDF) used in the protocol execution of the test session. Specifically, we initialize a $\mathsf{prf}^{\mathsf{swap}}$ challenger and query $\texttt{shk3}$, and use the output $\widetilde{C_8}$ from the $\mathsf{prf}^{\mathsf{swap}}$ challenger to replace the computation of C_8. Since by *Game 5b*, $\texttt{shk3}$ is a uniformly random and independent value, this replacement is sound. If the test bit sampled by the prf challenger is 0, then $\widetilde{C_8} \leftarrow \mathsf{KDF}(C_7, \texttt{shk3})$ and we are in *Game 5b*. If the test bit sampled by the prf challenger is 1, then $\widetilde{C_8} \xleftarrow{\$} \{0,1\}^{|\mathsf{KDF}|}$ and we are in *Game 5c*.

Thus any adversary capable of distinguishing this change can be turned into a successful adversary against the $\mathsf{prf}^{\mathsf{swap}}$ security of KDF, and we find:

$$\Pr[\text{break}_{5b}] \leq \mathsf{Adv}^{\mathsf{prf}^{\mathsf{swap}}}_{\mathsf{KDF},\mathcal{R}}(\lambda) + \Pr[\text{break}_{5c}]$$

Game 6 In this game we replace the values C_9, tmp, κ_9 with uniformly random and independent values $\widetilde{C_9}, \widetilde{tmp}, \widetilde{\kappa_9} \xleftarrow{\$} \{0,1\}^{|\mathsf{KDF}|}$ (where $\{0,1\}^{|\mathsf{KDF}|}$ is the output space of KDF) used in the protocol execution of the test session. Specifically, we initialize a PRF challenger and issue the challenge psk to it, and use the output $\widetilde{C_9}, \widetilde{tmp}, \widetilde{\kappa_9}$ from the PRF challenger to replace the computation of C_9, tmp, κ_9. Since by *Game 5c*, $\widetilde{C_8}$ is a uniformly random and independent value, this replacement is sound. If the test bit sampled by the prf challenger is 0, then $\widetilde{C_9}, \widetilde{tmp}, \widetilde{\kappa_4} \leftarrow \mathsf{KDF}(C_8, psk)$ and we are in *Game 5c*. If the test bit sampled by the prf challenger is 1, then $\widetilde{C_9}, tmp, \widetilde{\kappa_9} \xleftarrow{\$} \{0,1\}^{|\mathsf{KDF}|}$ and we are in *Game 6*. Thus any adversary $\mathcal{A}$ capable of distinguishing this

change can be turned into a successful adversary against the PRF assumption, and we find:

$$\Pr[\text{break}_{5_c}] \leq \mathsf{Adv}^{\mathsf{prf}}_{\mathsf{KDF},\mathcal{R}}(\lambda) + \Pr[\text{break}_6]$$

Game 7 In this game we replace the values $C_{10}, \kappa_{10} \leftarrow \mathsf{KDF}(\widetilde{C_9}, \emptyset)$ with uniformly random and independent values $\widetilde{C_{10}}, \widetilde{\kappa_{10}} \xleftarrow{\$} \{0,1\}^{|\mathsf{KDF}|}$ (where $\{0,1\}^{|\mathsf{KDF}|}$ is the output space of the KDF) used in the protocol execution of the test session. Specifically, we initialize a PRF challenger and issue the challenge query $\emptyset$ to it, and use the output $\widetilde{C_{10}}, \widetilde{\kappa_{10}}$ from the prf challenger to replace the computation of C_{10}, κ_{10}. Since by *Game 6*, $\widetilde{C_9}$ is a uniformly random and independent value, this replacement is sound. If the test bit sampled by the PRF challenger is 0, then $\widetilde{C_{10}}, \widetilde{\kappa_{10}} \leftarrow \mathsf{KDF}(\widetilde{C_9}, \emptyset)$ and we are in *Game 6*. If the test bit sampled by the prf challenger is 1, then $\widetilde{C_{10}}, \widetilde{\kappa_{10}} \xleftarrow{\$} \{0,1\}^{|\mathsf{KDF}|}$ and we are in *Game 7*. Thus any adversary $\mathcal{A}$ capable of distinguishing this change can be turned into a successful adversary against the prf assumption, and we find:

$$\Pr[\text{break}_6] \leq \mathsf{Adv}^{\mathsf{prf}}_{\mathsf{KDF},\mathcal{R}}(\lambda) + \Pr[\text{break}_7]$$

Game 8 In this game, we add an abort event $abort_{\mathsf{decrypt}}$ that triggers if the test session π_i^s receives a ciphertext `conf` in the `SenderConf` message that decrypts correctly. Since the test session π_i^s will only reach an accept status if `conf` decrypts correctly, it follows that

$$\Pr[\text{break}_7] \leq \Pr(abort_{\mathsf{decrypt}}).$$

Now we show that the probability of $abort_{\mathsf{decrypt}}$ is negligibly close to zero. We do so by initializing an aead-au challenger to decrypt `SenderConf.conf` ciphertexts in the test session π_i^s. We note that by *Game 7*, $\widetilde{\kappa_9}$ is a uniformly random and independent value, and since the aead challenger samples the internal aead key from the same distribution thus this change is undetectable. If π_i^s receives a ciphertext `conf` in the `SenderConf` message that decrypts correctly and the aead encryption oracle has not been queried, then it follows that this ciphertext `conf` is a forged ciphertext, breaking the au security of the AEAD scheme. Thus, we find that:

$$\Pr(abort_{\mathsf{decrypt}}) \leq \mathsf{Adv}^{\mathsf{auth\text{-}aead}}_{\mathsf{AEAD},\mathcal{R}}(\lambda).$$

Thus we find that the probability of $\mathcal{A}$ in causing a session π_i^s with $\rho = \texttt{resp}$ to reach $\pi_i^s.\alpha \leftarrow \texttt{accept}$ and triggering $\Pr\left[\text{break}_{\texttt{accept}}\right]$ to be:

$$\begin{aligned}\Pr(abort_{\texttt{accept}}) \leq (&\frac{n_S}{2^\lambda} + \mathsf{Adv}^{\text{IND-CCA}}_{\text{CCAKEM},\mathcal{R}}(\lambda) \\ &+ \ 3 \cdot \mathsf{Adv}^{\text{prf}}_{\text{KDF},\mathcal{R}}(\lambda) \\ &+ \ 2 \cdot \mathsf{Adv}^{\text{prf}^{\text{swap}}}_{\text{KDF},\mathcal{R}}(\lambda) \\ &+ \mathsf{Adv}^{\text{auth-aead}}_{\text{AEAD},\mathcal{R}}(\lambda)).\end{aligned}$$

We can finally show that

$$\begin{aligned}\mathsf{Adv}^{\text{eCK-PFS-PSK},C_2}_{\text{pqWG},n_P,n_S,\mathcal{A}}(\lambda) \leq n_P^2 \cdot n_S\Big(&\frac{n_S}{2^\lambda} + \mathsf{Adv}^{\text{IND-CCA}}_{\text{CCAKEM},\mathcal{R}}(\lambda) \\ &+ 3 \cdot \mathsf{Adv}^{\text{prf}}_{\text{KDF},\mathcal{R}}(\lambda) \\ &+ \ 2 \cdot \mathsf{Adv}^{\text{prf}^{\text{swap}}}_{\text{KDF},\mathcal{R}}(\lambda) \\ &+ \mathsf{Adv}^{\text{auth-aead}}_{\text{AEAD},\mathcal{R}}(\lambda)\Big).\end{aligned}$$

4.7.3. Case 3: Test session with contributive keyshare partner

By the case definition and the definition of the cleanness predicate $\mathsf{clean}_{\text{eCK-PFS-PSK}}$ there are five ways that the cleanness predicate could potentially be upheld[6]: $\mathcal{A}$ has issued $\mathsf{Test}(i,s)$ where $\mathsf{clean}_{\text{eCK-PFS-PSK}}(\pi_i^s)$ is upheld and has a contributive keyshare session π_j^t and either:

1. A pre-shared key exists between party P_i and the test session's intended partner, *and* $\mathcal{A}$ did not issue $\mathsf{CorruptPSK}(i,j)$, or $\mathsf{CorruptPSK}(j,i)$. We denote with $\mathsf{Adv}^{\text{eCK-PFS-PSK},C_{3.1}}_{\text{pqWG},n_P,n_S,\mathcal{A}}(\lambda)$ the advantage of $\mathcal{A}$ in winning in this case and refer to this as the *pre-shared subcase*.

2. $\mathcal{A}$ did not issue $\mathsf{CorruptEPK}(i,s)$ or $\mathsf{CorruptEPK}(j,t)$. We denote with $\mathsf{Adv}^{\text{eCK-PFS-PSK},C_{3.2}}_{\text{pqWG},n_P,n_S,\mathcal{A}}(\lambda)$ the advantage $\mathcal{A}$ and refer to this as the *ephemerals subcase*.

[6]Note that we do not make explicit in each condition that $\mathcal{A}$ has not issued either a $\mathsf{Reveal}(i,s)$ or $\mathsf{Reveal}(j,t)$ query

3. $\mathcal{A}$ did not issue $\mathsf{CorruptEPK}(i, s)$ or $\mathsf{CorruptASK}(j)$. We denote with $\mathsf{Adv}^{\mathsf{eCK\text{-}PFS\text{-}PSK},C_{3.3}}_{\mathsf{pqWG},n_P,n_S,\mathcal{A}}(\lambda)$ the advantage of $\mathcal{A}$ and refer to this as the *ephemeral/long-term subcase*.

4. $\mathcal{A}$ did not issue $\mathsf{CorruptASK}(i)$ or $\mathsf{CorruptEPK}(j, t)$. We denote with $\mathsf{Adv}^{\mathsf{eCK\text{-}PFS\text{-}PSK},C_{3.4}}_{\mathsf{pqWG},n_P,n_S,\mathcal{A}}(\lambda)$ the advantage of $\mathcal{A}$ and refer to this as the *long-term/ephemeral subcase*.

5. $\mathcal{A}$ did not issue $\mathsf{CorruptASK}(i)$ or $\mathsf{CorruptASK}(j)$. We denote with $\mathsf{Adv}^{\mathsf{eCK\text{-}PFS\text{-}PSK},C_{3.5}}_{\mathsf{pqWG},n_P,n_S,\mathcal{A}}(\lambda)$ the advantage of $\mathcal{A}$ and refer to this as the *long-terms subcase*.

Since at least one of these subcases must apply, then:

$$\mathsf{Adv}^{\mathsf{eCK\text{-}PFS\text{-}PSK},C_3}_{\mathsf{pqWG},n_P,n_S,\mathcal{A}}(\lambda) = \max\left\{\begin{array}{l}\mathsf{Adv}^{\mathsf{eCK\text{-}PFS\text{-}PSK},C_{3.1}}_{\mathsf{pqWG},n_P,n_S,\mathcal{A}}(\lambda),\\ \mathsf{Adv}^{\mathsf{eCK\text{-}PFS\text{-}PSK},C_{3.2}}_{\mathsf{pqWG},n_P,n_S,\mathcal{A}}(\lambda),\\ \mathsf{Adv}^{\mathsf{eCK\text{-}PFS\text{-}PSK},C_{3.3}}_{\mathsf{pqWG},n_P,n_S,\mathcal{A}}(\lambda),\\ \mathsf{Adv}^{\mathsf{eCK\text{-}PFS\text{-}PSK},C_{3.4}}_{\mathsf{pqWG},n_P,n_S,\mathcal{A}}(\lambda),\\ \mathsf{Adv}^{\mathsf{eCK\text{-}PFS\text{-}PSK},C_{3.5}}_{\mathsf{pqWG},n_P,n_S,\mathcal{A}}(\lambda)\end{array}\right\}$$

We now turn to bounding the advantage of the adversary $\mathcal{A}$ in each of the subcases, and show that if the advantage of $\mathcal{A}$ in each subcase is negligible, then so too is the advantage of $\mathcal{A}$ in Case 3.

4.7.4. Case 3.1: The Preshared Subcase

In this subcase we assume that the cleanness predicate is upheld such that a pre-shared secret between the test session and its honest contributive keyshare session exists, and has not been corrupted. Due to the definition of Case 3, we know that such an honest contributive keyshare session exists. In what follows, we show that the probability of $\mathcal{A}$ in winning the key-indistinguishability game is negligible.

Proof sketch We begin by guessing the index of the test session, and add an abort event that occurs if a Test query is directed to a session that does not have the index of the guessed session, and similarly, guess the index of the contributive keyshare partner. We then replace the value of C_9, tmp, κ_9 with uniformly random values $\widetilde{C_9}, \widetilde{tmp}, \widetilde{\kappa_9}$, and note that by the subcase definition and the $\mathsf{clean}_{\mathsf{eCK\text{-}PFS\text{-}PSK}}$, that the adversary cannot issue either a

$\mathsf{CorruptPSK}(i,j)$ or $\mathsf{CorruptPSK}(j,i)$ query. Since the psk shared between the two parties is a uniformly random and independent value, we argue that any adversary capable of distinguishing this replacement would be able to break the PRF assumption. In a similar fashion, we replace the values C_{10}, κ_{10} with uniformly random and independent values $\widetilde{C_{10}}, \widetilde{\kappa_{10}}$, and argue that since $\widetilde{C_9}$ was already independent from the protocol execution that this replacement was sound and that any adversary capable of distinguishing this change would be able to be turned into a adversary against PRF security. In the final game and with a similar argument, we replace tk_i, tk_r with uniformly random and independent values, based on the PRF security of KDF. Since the session keys are now uniformly random and independent of the test bit b sampled by the challenger, the advantage of $\mathcal{A}$ against the eCK-PFS-PSK-security of the modified WireGuard protocol in the pre-shared key subcase is negligible.

Game 0 This is a standard eCK-PFS-PSK with cleanness predicate $\mathsf{clean}_{\mathsf{eCK\text{-}PFS\text{-}PSK}}$ upheld as in Definition 53. Thus

$$\mathsf{Adv}^{\mathsf{eCK\text{-}PFS\text{-}PSK},C_{3.1}}_{\mathsf{pqWG},n_P,n_S,\mathcal{A}}(\lambda) = \Pr\left[\mathrm{break}_0\right].$$

Game 1 In this game, we guess the index (i,s) of the Test session π_i^s and abort if, during the experiment, a query $\mathsf{Test}(i^*,s^*)$ is issued such that $(i^*,s^*) \neq (i,s)$. Thus

$$\Pr\left[\mathrm{break}_0\right] \leq n_P \cdot n_S \cdot \Pr\left[\mathrm{break}_1\right].$$

Game 2 In this game, we guess the index (j,t) of the contributive keyshare session π_j^t (which exists by the Case 3 definition) and abort if during the experiment, a query $\mathsf{Test}(i,s)$ is issued when the contributive keyshare session $\pi_{t^*}^{j^*}$ exists such that $(j^*,t^*) \neq (j,t)$. Thus

$$\Pr\left[\mathrm{break}_1\right] \leq n_P \cdot n_S \cdot \Pr\left[\mathrm{break}_2\right].$$

Game 3 In this game, we replace the computation of C_9, tmp, κ_9 with uniformly random values $\widetilde{C_9}, \widetilde{tmp}, \widetilde{\kappa_9}$ in the execution of session π_i^s and its partner session π_j^t. We do so by interacting with a $\mathsf{prf}^{\mathsf{swap}}$ challenger in the following way: When it is time to compute $C_9, tmp, \kappa_9 \leftarrow \mathsf{KDF}(C_8, psk)$ we instead initialize a $\mathsf{prf}^{\mathsf{swap}}$ challenger and query C_8. We note that by the cleanness predicate and the preconditions of this subcase that psk is a uniformly random value that will not be revealed by $\mathcal{A}$ through a $\mathsf{CorruptPSK}(i,j)$ query, and thus this replacement is sound. If the random bit b sampled by the prf challenger is 0, then we are in *Game 2*. If the random bit b sampled by the

$\mathsf{prf}^{\mathsf{swap}}$ challenger is 1, then we are in *Game 3*. Any adversary $\mathcal{A}$ capable of distinguishing this change in the experiment can be turned into an algorithm against the $\mathsf{prf}^{\mathsf{swap}}$ security of KDF and thus

$$\Pr[\text{break}_2] \leq \mathsf{Adv}^{\mathsf{prf}^{\mathsf{swap}}}_{\mathsf{KDF},\mathcal{R}}(\lambda) + \Pr[\text{break}_3].$$

Game 4 Similarly to the previous game, we replace the computation of C_{10} with a uniformly random value $\widetilde{C_{10}}$ from the same distribution, in the challenger's execution of the test session π_i^s and its partner session π_j^t. We do so by interacting with a prf challenger in the following way: When it is time to compute $C_{10} \leftarrow \mathsf{KDF}(C_9, \emptyset)$ we instead initialize a prf challenger and query it with the empty string $\emptyset$. We note that by *Game 3*, C_9 is a uniformly random value independent from the protocol execution, and as such the replacement is sound.If the random bit b sampled by the prf challenger is 0, then we are in *Game 3*. If the random bit b sampled by the prf challenger is 1, then we are in *Game 4*. Any adversary $\mathcal{A}$ capable of distinguishing this change in the experiment can be turned into an algorithm against the prf security of KDF, and thus

$$\Pr[\text{break}_3] \leq \mathsf{Adv}^{\mathsf{prf}}_{\mathsf{prf},\mathcal{R}}(\lambda) + \Pr[\text{break}_4].$$

Game 5 As in previous games, we replace the values $tk_i, tk_r \leftarrow \mathsf{KDF}(\widetilde{C_{10}}, \emptyset)$ computed by the challenger in the execution of the test session and its honest contributive keyshare session partner π_j^t with uniformly random values $\widetilde{tk_i}, \widetilde{tk_r}$. We do so by interacting with a prf challenger in the following way: When it is time to compute tk_i, tk_r in the appropriate sessions, we instead initialize a prf challenger and query it with the empty string $\emptyset$. We note that by *Game 4*, $\widetilde{C_{10}}$ is a uniformly random value independent from the protocol execution, and as such the replacement is sound.If the random bit sampled by the prf challenger if 0, then we are in *Game 4*, but otherwise the output of the prf challenger $\widetilde{tk_i}, \widetilde{tk_r}$ is uniformly random and independent and we are in *Game 5*. Any adversary $\mathcal{A}$ capable of distinguishing this change in the experiment can be turned into an algorithm against the prf security of KDF, and thus

$$\Pr[\text{break}_4] \leq \mathsf{Adv}^{\mathsf{prf}}_{\mathsf{KDF},\mathcal{R}}(\lambda) + \Pr[\text{break}_5].$$

Since the response to the $\mathsf{Test}(i, s)$ query is (in *Game 5*) uniformly random and independent regardless of the value of the test bit b, then the adversary's success in winning the key-indistinguishability game is reduced to simply guessing and thus:

$$\mathsf{Adv}^{\mathsf{eCK\text{-}PFS\text{-}PSK},C_{3.1}}_{\mathsf{pqWG},n_P,n_S,\mathcal{R}}(\lambda) \leq n_P^2 \cdot n_S^2 \Big(\quad 2 \cdot \mathsf{Adv}^{\mathsf{prf}}_{\mathsf{KDF},\mathcal{R}}(\lambda) + \mathsf{Adv}^{\mathsf{prf}^{\mathsf{swap}}}_{\mathsf{KDF},\mathcal{R}}(\lambda) \Big).$$

4.7.5. Case 3.2: The Ephemerals Subcase

In this subcase we know that (by the definition of $\mathsf{clean}_{\mathsf{eCK\text{-}PFS\text{-}PSK}}$ and the subcase preconditions) that the session π_i^s such that the $\mathsf{Test}(i,s)$ session will be queried has an honest contributive keyshare session π_j^t and that $\mathsf{CorruptEPK}(i,s)$ and $\mathsf{CorruptEPK}(j,t)$ queries have not been issued during the execution of the experiment. We now show that in this subcase, the adversary's probability in winning the key-indistinguishability game is negligible.

Game 0 This is a standard eCK-PFS-PSK game with cleanness predicate $\mathsf{clean}_{\mathsf{eCK\text{-}PFS\text{-}PSK}}$ upheld. Thus we have

$$\mathsf{Adv}^{\mathsf{eCK\text{-}PFS\text{-}PSK}}_{\mathsf{pqWG},n_P,n_S,\mathcal{A}}(\lambda) = \Pr\left[\mathrm{break}_0\right].$$

Game 1 In this game, we guess the index (i,s) of the Test session π_i^s and abort if, during the experiment, a query $\mathsf{Test}(i^*,s^*)$ is issued such that $(i^*,s^*) \neq (i,s)$. Thus

$$\Pr\left[\mathrm{break}_0\right] \leq n_P \cdot n_S \cdot \Pr\left[\mathrm{break}_1\right].$$

Game 2 In this game, we guess the index (j,t) of the honest partner session π_j^t (which we know exists by the Case 3 definition) and abort if, during the experiment, a query $\mathsf{Test}(i,s)$ is issued if the contributive keyshare session $\pi_{t^*}^{j^*}$ exists such that $(j^*,t^*) \neq (i,s)$. Thus

$$\Pr\left[\mathrm{break}_1\right] \leq n_P \cdot n_S \cdot \Pr\left[\mathrm{break}_2\right].$$

Game 3 is somewhat special in that both ephemeral keys are assumed to be uncorrupted. In the original version this meant that only the DDH-assumption was necessary, whereas our version only requires an IND-CPA-secure KEM. We again follow the original proof as closely as possible:

In this game, we replace the value `ct2` computed in the test session π_i^s and its honest contributive keyshare session with a random element from the same keyspace.

This works because we know from the definition of Case 3 that there is a contributive keyshare session, and we know from condition 5 of the freshness-predicate that matching sessions agree on their key-material. Therefore the ciphertext that the initiator receives for the IND-CPA-secure KEM has to be the honestly generated one in this case and we can replace it as receiving any adversarially generated ciphertext would violate the freshness-predicate of Case 3.2. Lastly we know from *Game 3.1* that the encapsulated keys match and that there is no decryption failure that our substitution would remove.

We explicitly interact with an ICC-CPA challenger, and replace the ephemeral epk_i and $\texttt{ct2}$ values sent in the InitiatorHello and ResponderHello messages with the challenge public-key and ciphertext from the ICC-CPA challenger. We only require the encapsulated key in one computation (as opposed to three in the original proof):

- $C_7 \leftarrow \mathsf{KDF}(c_2, \texttt{shk2})$

Here we can replace $\texttt{shk2}$ with the supposed shared key $\texttt{k}^*$ from the ICC-CPA-challenger. When the test bit sampled by the ICC-CPA challenger is 0, then $\texttt{k}^*$ is the actually encapsulated shared key and we are in *Game 2*. When the test bit sampled by the ICC-CPA challenger is 1, then $\texttt{k}^* \xleftarrow{\$} \mathcal{K}_{\mathsf{CPAKEM}}$ and we are in *Game 3*. Any adversary that can detect that change can be turned into an adversary against the IND-CPA problem and thus

$$\Pr[\text{break}_2] \leq \mathsf{Adv}^{\mathsf{ICC\text{-}CPA}}_{\mathsf{CPAKEM},\mathcal{R}}(\lambda) + \Pr[\text{break}_3].$$

By applying Lemma 2, we get:

$$\Pr[\text{break}_2] \leq \mathsf{Adv}^{\mathsf{IND\text{-}CPA}}_{\mathsf{CPAKEM},\mathcal{R}}(\lambda) + \mathsf{CPAKEM}.\delta + \Pr[\text{break}_3].$$

Game 4 In this game, we replace the computation of C_7 with a uniformly random value $\widetilde{C_7}$ from the same distribution, in the challenger's execution of the test session π_i^s and its partner session π_j^t. We do so by interacting with a $\mathsf{prf}^{\mathsf{swap}}$ challenger in the following way: When it is time to compute $C_7 \leftarrow \mathsf{KDF}(C_6, \texttt{shk2})$ we instead initialize a KDF challenger and query it with C_6. We note that by *Game 3*, $\texttt{shk2}$ is a uniformly random and independent value, and thus this replacement is sound. If the random bit b sampled by the $\mathsf{prf}^{\mathsf{swap}}$ challenger is 0, then $\widetilde{C_7} \leftarrow \mathsf{KDF}(C_6, \texttt{shk2})$ and we are in *Game 3*. If the random bit b sampled by the prf challenger is 1, then $\widetilde{C_7} \xleftarrow{\$} \{0,1\}^{|\mathsf{KDF}|}$ and we are in *Game 4*. Any adversary $\mathcal{A}$ capable of distinguishing this change

in the experiment can be turned into an algorithm against the prf security of KDF and thus

$$\Pr[\text{break}_3] \leq \mathsf{Adv}^{\mathsf{prf}^{\mathsf{swap}}}_{\mathsf{KDF},\mathcal{R}}(\lambda) + \Pr[\text{break}_4].$$

Game 5 Similarly to the previous game, we replace the computation of C_8 with a uniformly random value $\widetilde{C_8}$ from the same distribution, in the challenger's execution of the test session π_i^s and its partner session π_j^t. We do so by interacting with a prf challenger in the following way: When it is time to compute $C_8 \leftarrow \mathsf{KDF}(\widetilde{C_7}, \texttt{shk3})$ we instead initialize a prf challenger and query it with shk3. We note that by *Game 4*, $\widetilde{C_7}$ is a uniformly random and independent value, and thus this replacement is sound. If the random bit b sampled by the prf challenger is 0, then $C_8 \leftarrow \mathsf{KDF}(\widetilde{C_7}, \texttt{shk3})$ and we are in *Game 4*. If the random bit b sampled by the prf challenger is 1, then $\widetilde{C_8} \xleftarrow{\$} \{0,1\}^{|\mathsf{KDF}|}$ and we are in *Game 5*. Any adversary $\mathcal{A}$ capable of distinguishing this change in the experiment can be turned into an algorithm against the prf security of KDF, and thus

$$\Pr[\text{break}_4] \leq \mathsf{Adv}^{\mathsf{prf}}_{\mathsf{KDF},\mathcal{R}}(\lambda) + \Pr[\text{break}_5].$$

Game 6 As in previous games, we replace the computation of C_9, tmp, κ_9 with uniformly random values $\widetilde{C_9}, \widetilde{tmp}, \widetilde{\kappa_9}$ from the same distribution, in the challenger's execution of the test session π_i^s and its partner session π_j^t. We do so by interacting with a prf challenger in the following way: When it is time to compute $C_9, tmp, \kappa_9 \leftarrow \mathsf{KDF}(\widetilde{C_8}, psk)$ we instead initialize a prf challenger and query it with psk. We note that by *Game 5*, $\widetilde{C_8}$ is a u,mly random value and independent value, and thus this replacement is sound. If the random bit b sampled by the prf challenger is 0, then we are in *Game 5*. If the random bit b sampled by the prf challenger is 1, then we are in *Game 6*. Any adversary $\mathcal{A}$ capable of distinguishing this change in the experiment can be turned into an algorithm against the prf security of KDF, and thus

$$\Pr[\text{break}_5] \leq \mathsf{Adv}^{\mathsf{prf}}_{\mathsf{KDF},\mathcal{R}}(\lambda) + \Pr[\text{break}_6].$$

Game 7 As in previous games, we replace the computation of C_{10} with a uniformly random value $\widetilde{C_{10}}$ from the same distribution, in the challenger's execution of the test session π_i^s and its partner session π_j^t. We do so by interacting with a prf challenger in the following way: When it is time to compute $C_{10} \leftarrow \mathsf{KDF}(C_9, \emptyset)$ we instead initialize a prf challenger and query it with the empty string $\emptyset$. We note that by *Game 6*, $\widetilde{C_9}$ is a uniformly random

value independent from the protocol execution, and as such the replacement is sound. If the random bit b sampled by the prf challenger is 0, then we are in *Game 6*. If the random bit b sampled by the prf challenger is 1, then we are in *Game 7*. Any adversary $\mathcal{A}$ capable of distinguishing this change in the experiment can be turned into an algorithm against the prf security of KDF, and thus

$$\Pr[\text{break}_6] \leq \mathsf{Adv}^{\mathsf{prf}}_{\mathsf{KDF},\mathcal{R}}(\lambda) + \Pr[\text{break}_7]$$

Game 8 As in previous games, we replace the values $tk_i, tk_r \leftarrow \mathsf{KDF}(\widetilde{C_{10}}, \emptyset)$ computed by the challenger in the execution of the test session and its honest contributive keyshare session partner π_j^t with uniformly random values $\widetilde{tk_i}, \widetilde{tk_r}$. We do so by interacting with a prf challenger in the following way: When it is time to compute tk_i, tk_r in the appropriate sessions, we instead initialize a prf challenger and query it with the empty string $\emptyset$. We note that by *Game 4*, $\widetilde{C_{10}}$ is a uniformly random value independent from the protocol execution, and as such the replacement is sound. If the random bit sampled by the prf challenger if 0, then we are in *Game 7*, but otherwise the output of the prf challenger $\widetilde{tk_i}, \widetilde{tk_r}$ is uniformly random and independent and we are in *Game 8*. Any adversary $\mathcal{A}$ capable of distinguishing this change in the experiment can be turned into an algorithm against the prf security of KDF, and thus

$$\Pr[\text{break}_7] \leq \mathsf{Adv}^{\mathsf{prf}}_{\mathsf{KDF},\mathcal{R}}(\lambda) + \Pr[\text{break}_8]$$

Since the response to the $\mathsf{Test}(i, s)$ query issued by the adversary is, in *Game 8*, uniformly random and independent of the test bit b sampled by the challenger, then the adversary's success in winning the key-indistinguishability game is reduced to simply guessing and thus:

$$\begin{aligned}\mathsf{Adv}^{\mathsf{eCK\text{-}PFS\text{-}PSK},C_{3.2}}_{\mathsf{pqWG},n_P,n_S,\mathcal{A}}(\lambda) \leq\ & n_P^2 \cdot n_S^2 \Big(\mathsf{CPAKEM}.\delta \\ & + \mathsf{Adv}^{\mathsf{IND\text{-}CPA}}_{\mathsf{CPAKEM},\mathcal{R}}(\lambda) \\ & + 4 \cdot \mathsf{Adv}^{\mathsf{prf}}_{\mathsf{KDF},\mathcal{R}}(\lambda) \\ & + \mathsf{Adv}^{\mathsf{prf}^{\mathsf{swap}}}_{\mathsf{KDF},\mathcal{R}}(\lambda) \Big)\end{aligned}$$

4.7.6. Case 3.3: The Ephemeral/Long-term Subcase

In this subcase we know that (by the definition of $\mathsf{clean}_{\mathsf{eCK\text{-}PFS\text{-}PSK}}$ and the subcase preconditions) that the session π_i^s such that the $\mathsf{Test}(i, s)$ session will be queried has an honest contributive keyshare session π_j^t and that

$\mathsf{CorruptEPK}(i,s)$ and $\mathsf{CorruptASK}(j)$ queries have not been issued during the execution of the experiment. Note that in our proof we set that the test session has role `init` and the partner session has role `resp`, but the case where the test session has role `resp` and the partner session has role `init` follows analogously. In what follows, we show that in this subcase, the adversary's probability in winning the key-indistinguishability game is negligible under certain security assumptions.

Game 0 This is a standard eCK-PFS-PSK game with cleanness predicate $\mathsf{clean}_{\mathsf{eCK\text{-}PFS\text{-}PSK}}$ upheld. Thus we have:

$$\mathsf{Adv}^{\mathsf{eCK\text{-}PFS\text{-}PSK}}_{\mathsf{pqWG},n_P,n_S,\mathcal{A}}(\lambda) = \Pr\left[\mathrm{break}_0\right]$$

Game 1 In this game, we guess the index (i,s) of the Test session π_i^s and abort if, during the experiment, a query $\mathsf{Test}(i^*,s^*)$ is issued such that $(i^*,s^*) \neq (i,s)$. Thus:

$$\Pr\left[\mathrm{break}_0\right] \leq n_P \cdot n_S \cdot (\Pr\left[\mathrm{break}_1\right])$$

Game 2 In this game, we guess the index (j,t) of the honest partner session π_j^t (which we know exists by the Case 3 definition) and abort if, during the experiment, a query $\mathsf{Test}(i,s)$ is issued if the contributive keyshare session $\pi_{t^*}^{j^*}$ exists such that $(j^*,t^*) \neq (i,s)$. Thus:

$$\Pr\left[\mathrm{break}_1\right] \leq n_P \cdot n_S \cdot (\Pr\left[\mathrm{break}_2\right])$$

Game 3 In this game we replace the computation of C_3, κ_3 with uniformly random and independent values $\widetilde{C_3}, \widetilde{\kappa_3}$. This is practically identical to the *Game 5* of case 1, including the subhybrids.

In ***Game 3a*** we replace the value $\hat{r} := \mathsf{KDF}(\sigma_i, r_i)$ passed to $\mathsf{CCAKEM.Enc}$ for the computation of `ct1` and `shk1` with a random bitstring $\hat{r}'$.

To show that this replacement is sound, we replace the value of $\hat{r}$ with a uniformly random and independent value $\hat{r}' \xleftarrow{\$} \{0,1\}^{|\mathsf{KDF}|}$ used in the protocol execution of the test session. Specifically, we initialize a $\mathsf{prf}^{\mathsf{swap}}$ challenger and query σ_i, and use the output $\tilde{r}$ from the $\mathsf{prf}^{\mathsf{swap}}$ challenger to replace the computation of $\hat{r}$. By the definition of this case r_i is a uniformly random and independent value, therefore this replacement is sound. If the test bit sampled by the $\mathsf{prf}^{\mathsf{swap}}$ challenger is 0, then $\hat{r} \leftarrow \mathsf{KDF}(\sigma_i, r_i)$ and we are in *Game 2*. If the test bit sampled by the $\mathsf{prf}^{\mathsf{swap}}$ challenger is 1, then $\hat{r} \xleftarrow{\$} \{0,1\}^{|\mathsf{KDF}|}$ is a truly random value and we are in *Game 3a*.

Thus any adversary capable of distinguishing this change can be turned into a successful adversary against the $\mathsf{prf}^{\mathsf{swap}}$ security of KDF, and we find:

$$\Pr[\text{break}_2] \leq \mathsf{Adv}^{\mathsf{prf}^{\mathsf{swap}}}_{\mathsf{KDF},\mathcal{R}}(\lambda) + \Pr[\text{break}_{3a}]$$

In ***Game 3b*** we replace the computation of `shk1` by sampling the value uniformly at random from the space of shared secrets of the KEM and ignoring the second output of $\mathsf{CCAKEM.Enc}(\mathtt{spk}_r)$. To show that this is undetectable under the IND-CCA-assumption of the used KEM, we interact with an IND-CCA challenger in the following way: Note that by *Game 2*, we know at the beginning of the experiment the index of session π_i^s such that $\mathsf{Test}(i,s)$ is issued by the adversary. Similarly, by *Game 1*, we know at the beginning of the experiment the index of the intended partner P_j of the session π_i^s. Thus, we initialize an IND-CCA challenger and use the received public-key $\mathtt{pk}^*$ as long-term public-key of party P_j and give it with all other (honestly generated) public keys to the adversary. Note that by the definition of this case, $\mathcal{A}$ is not able to issue a $\mathsf{CorruptASK}(j)$ query, as we abort if $\pi_i^s.\alpha \leftarrow \mathtt{reject}$ and abort if $\pi_i^s.\alpha \leftarrow \mathtt{accept}$. Thus we will not need to reveal the private key $\mathtt{sk}^*$ of the challenge public-key to $\mathcal{A}$. However we must account for all sessions t such that π_j^t must use the private key for computations. In our version of WireGuard, the long-term private keys are used to compute the following:

- In sessions where P_j acts as the initiator:
  $C_8 \leftarrow \mathsf{KDF}(C_6, \mathsf{CCAKEM.Dec}(\mathtt{ssk}_i, \mathtt{ct3}))$
- In sessions where P_j acts as the responder:
  $C_3, \kappa_3 \leftarrow \mathsf{KDF}(C_2, \mathsf{CCAKEM.Dec}(\mathtt{ssk}_r, \mathtt{ct1}))$

(Note that these are fewer cases than in the original proof because we don't combine static and ephemeral keys directly.) Dealing with the challenger's computation of these values will be done in two ways:

- The encapsulation was created by another honest party. The challenger can then use its own internal knowledge of the encapsulated value to complete the computations.
- The encapsulation was not created by another honest party, but by the adversary and the challenger is therefore unaware of the encapsulated value.

In the second case, the challenger can instead use the decapsulation-oracle provided by the CCA-challenger, specifically querying $\mathsf{CCAKEM.Dec}(\mathtt{ctX})$,

(where $\mathtt{ctX}$ is the relevant encapsulation) which will output $\mathtt{shkX}$ using the CCA challenger's internal knowledge of $\mathtt{sk}^*$.

During session i we request a challenge consisting of a ciphertext and a candidate shared secret $(\mathtt{c}^*, \mathtt{k}^*)$ from the IND-CCA challenger and use those values in place of $\mathtt{ct1}$ and $\mathtt{shk1}$. Given the definition of the IND-CCA game, there are two cases:

- If the test bit sampled by the IND-CCA challenger is 0, then $\mathtt{k}^*$ is indeed the shared secret encapsulated in $\mathtt{c}^*$ and we are in *Game 3a*.
- If the test bit sampled by the IND-CCA challenger is 1, then $\mathtt{k}^*$ is not the shared secret encapsulated in $\mathtt{c}^*$ but sampled uniformly at random from the space of shared secrets and we are in *Game 3b*.

Thus, any adversary capable of distinguishing this change can be turned into a successful adversary against the IND-CCA security of the used KEM and we find:

$$\Pr\left[\text{break}_{3a}\right] \leq \mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathsf{CCAKEM},\mathcal{R}}(\lambda) + \Pr\left[\text{break}_{3b}\right]$$

In ***Game 3c*** we replace the values of C_3, κ_3 with uniformly random and independent values $\widetilde{C_3}, \widetilde{\kappa_3} \xleftarrow{\$} \{0,1\}^{|\mathsf{KDF}|}$ (where $\{0,1\}^{|\mathsf{KDF}|}$ is the output space of the KDF) used in the protocol execution of the test session. Specifically, we initialize a $\mathsf{prf}^{\mathsf{swap}}$ challenger and query $\mathtt{shk1}$, and use the output $\widetilde{C_3}, \widetilde{\kappa_3}$ from the $\mathsf{prf}^{\mathsf{swap}}$ challenger to replace the computation of C_3, κ_3. Since by *Game 3b*, $\mathtt{shk1}$ is a uniformly random and independent value, this replacement is sound. If the test bit sampled by the $\mathsf{prf}^{\mathsf{swap}}$ challenger is 0, then $\widetilde{C_3}, \widetilde{\kappa_3} \leftarrow \mathsf{KDF}(C_2, \mathtt{shk1})$ and we are in *Game 3b*. If the test bit sampled by the $\mathsf{prf}^{\mathsf{swap}}$ challenger is 1, then $\widetilde{C_3}, \widetilde{\kappa_3} \xleftarrow{\$} \{0,1\}^{|\mathsf{KDF}|}$ and we are in *Game 3c*.

Thus any adversary capable of distinguishing this change can be turned into a successful adversary against the $\mathsf{prf}^{\mathsf{swap}}$ security of KDF, and we find:

$$\Pr\left[\text{break}_{3b}\right] \leq \mathsf{Adv}^{\mathsf{prf}^{\mathsf{swap}}}_{\mathsf{KDF},\mathcal{R}}(\lambda) + \Pr\left[\text{break}_{3c}\right]$$

Game 4 In this game, we replace the computation of C_4, κ_4 with uniformly random values $\widetilde{C_4}, \widetilde{\kappa_4}$ from the same distribution, in the challenger's execution of the test session π_i^s and its partner session π_j^t. We do so by interacting with a prf challenger in the following way: When it is time to compute $C_4, \kappa_4 \leftarrow \mathsf{KDF}(\widetilde{C_3}, psk)$ we instead initialize a prf challenger and query it with psk. We note that by *Game 3c* that $\widetilde{C_3}$ is a uniformly random and independent value,

and thus this replacement is sound. If the random bit b sampled by the prf challenger is 0, then we are in *Game 3c*. If the random bit b sampled by the prf challenger is 1, then we are in *Game 4*. Any adversary $\mathcal{A}$ capable of distinguishing this change in the experiment can be turned into an algorithm against the prf security of KDF and thus:

$$\Pr[\text{break}_{3c}] \leq \mathsf{Adv}^{\mathsf{prf}}_{\mathsf{KDF},\mathcal{R}}(\lambda) + \Pr[\text{break}_4]$$

Game 5 In this game, we replace the computation of C_6 with a uniformly random value $\widetilde{C_6}$ from the same distribution, in the challenger's execution of the test session π_i^s and its partner session π_j^t. We do so by interacting with a prf challenger in the following way: When it is time to compute $C_6 \leftarrow \mathsf{KDF}(\widetilde{C_4}, \texttt{ct2})$ we instead initialize a prf challenger and query it with ct2. We note that by *Game 4*, $\widetilde{C_4}$ is a uniformly random and independent value, and thus this replacement is sound. If the random bit b sampled by the prf challenger is 0, then we are in *Game 4*. If the random bit b sampled by the prf challenger is 1, then we are in *Game 5*. Any adversary $\mathcal{A}$ capable of distinguishing this change in the experiment can be turned into an algorithm against the prf security of KDF and thus:

$$\Pr[\text{break}_4] \leq \mathsf{Adv}^{\mathsf{prf}}_{\mathsf{KDF},\mathcal{R}}(\lambda) + \Pr[\text{break}_5]$$

Game 6 In this game, we replace the computation of C_7 with a uniformly random value $\widetilde{C_7}$ from the same distribution, in the challenger's execution of the test session π_i^s and its partner session π_j^t. We do so by interacting with a prf challenger in the following way: When it is time to compute $C_7 \leftarrow \mathsf{KDF}(\widetilde{C_6}, \texttt{shk2})$ we instead initialize a prf challenger and query it with shk2. We note that by *Game 5*, $\widetilde{C_6}$ is a uniformly random and independent value, and thus this replacement is sound. If the random bit b sampled by the prf challenger is 0, then $\widetilde{C_7} \leftarrow \mathsf{KDF}(\widetilde{C_6}, \texttt{shk2})$ and we are in *Game 5*. If the random bit b sampled by the prf challenger is 1, then $\widetilde{C_7} \xleftarrow{\$} \{0,1\}^{|\mathsf{KDF}|}$ and we are in *Game 6*. Any adversary $\mathcal{A}$ capable of distinguishing this change in the experiment can be turned into an algorithm against the prf security of KDF and thus:

$$\Pr[\text{break}_5] \leq \mathsf{Adv}^{\mathsf{prf}}_{\mathsf{KDF},\mathcal{R}}(\lambda) + \Pr[\text{break}_6]$$

Game 7 Similarly to the previous game, we replace the computation of C_8 with a uniformly random value $\widetilde{C_8}$ from the same distribution, in the challenger's execution of the test session π_i^s and its partner session π_j^t. We

do so by interacting with a prf challenger in the following way: When it is time to compute $C_8 \leftarrow \mathsf{KDF}(\widetilde{C_7}, \texttt{shk3})$ we instead initialize a prf challenger and query it with shk3. We note that by *Game 6*, $\widetilde{C_7}$ is a uniformly random and independent value, and thus this replacement is sound. If the random bit b sampled by the prf challenger is 0, then $C_8 \leftarrow \mathsf{KDF}(\widetilde{C_7}, \texttt{shk3})$ and we are in *Game 6*. If the random bit b sampled by the prf challenger is 1, then $\widetilde{C_8} \xleftarrow{\$} \{0,1\}^{|\mathsf{KDF}|}$ and we are in *Game 7*. Any adversary $\mathcal{A}$ capable of distinguishing this change in the experiment can be turned into an algorithm against the prf security of KDF and thus:

$$\Pr[\text{break}_6] \leq \mathsf{Adv}^{\mathsf{prf}}_{\mathsf{KDF},\mathcal{R}}(\lambda) + \Pr[\text{break}_7]$$

Game 8 In this game, we replace the computation of C_9, tmp, κ_9 with uniformly random values $\widetilde{C_9}, \widetilde{tmp}, \widetilde{\kappa_9}$ from the same distribution, in the challenger's execution of the test session π_i^s and its partner session π_j^t. We do so by interacting with a prf challenger in the following way: When it is time to compute $C_9, tmp, \kappa_9 \leftarrow \mathsf{KDF}(\widetilde{C_8}, psk)$ we instead initialize a prf challenger and query it with psk. We note that by *Game 7*, $\widetilde{C_8}$ is a uniformly random and independent value, and thus this replacement is sound. If the random bit b sampled by the prf challenger is 0, then we are in *Game 7*. If the random bit b sampled by the prf challenger is 1, then we are in *Game 8*. Any adversary $\mathcal{A}$ capable of distinguishing this change in the experiment can be turned into an algorithm against the prf security of KDF and thus:

$$\Pr[\text{break}_7] \leq \mathsf{Adv}^{\mathsf{prf}}_{\mathsf{KDF},\mathcal{R}}(\lambda) + \Pr[\text{break}_8]$$

Game 9 Similarly to the previous game, we replace the computation of C_{10} with a uniformly random value $\widetilde{C_{10}}$ from the same distribution, in the challenger's execution of the test session π_i^s and its partner session π_j^t. We do so by interacting with a prf challenger in the following way: When it is time to compute $C_{10} \leftarrow \mathsf{KDF}(C_9, \emptyset)$ we instead initialize a prf challenger and query it with the empty string $\emptyset$. We note that by *Game 8*, $\widetilde{C_9}$ is a uniformly random value independent from the protocol execution, and as such the replacement is sound If the random bit b sampled by the prf challenger is 0, then we are in *Game 8*. If the random bit b sampled by the prf challenger is 1, then we are in *Game 9*. Any adversary $\mathcal{A}$ capable of distinguishing this change in the experiment can be turned into an algorithm against the prf security of KDF and thus:

$$\Pr[\text{break}_8] \leq \mathsf{Adv}^{\mathsf{prf}}_{\mathsf{KDF},\mathcal{R}}(\lambda) + \Pr[\text{break}_9]$$

Game 10 Similarly to the previous games, we replace the values $tk_i, tk_r \leftarrow \mathsf{KDF}(\widetilde{C_{10}}, \emptyset)$ computed by the challenger in the execution of the test session and its honest contributive keyshare session partner π_j^t with uniformly random values $\widetilde{tk_i}, \widetilde{tk_r}$. We do so by interacting with a prf challenger in the following way: When it is time to compute tk_i, tk_r in the appropriate sessions, we instead initialize a prf challenger and query it with the empty string $\emptyset$. We note that by *Game 9*, $\widetilde{C_{10}}$ is a uniformly random value independent from the protocol execution, and as such the replacement is sound. If the random bit sampled by the prf challenger if 0, then we are in *Game 9*, but otherwise the output of the prf challenger $\widetilde{tk_i}, \widetilde{tk_r}$ is uniformly random and independent and we are in *Game 10*. Any adversary $\mathcal{A}$ capable of distinguishing this change in the experiment can be turned into an algorithm against the prf security of KDF and thus:

$$\Pr[\text{break}_9] \leq \mathsf{Adv}^{\mathsf{prf}}_{\mathsf{KDF},\mathcal{R}}(\lambda) + \Pr[\text{break}_{10}]$$

Since the response to the $\mathsf{Test}(i, s)$ query issued by the adversary is, in *Game 10*, uniformly random and independent of the test bit b sampled by the challenger, then the adversary's success in winning the key-indistinguishability game is reduced to simply guessing and thus:

$$\Pr[\text{break}_{10}] = 0$$

$$\begin{aligned}\mathsf{Adv}^{\mathsf{eCK\text{-}PFS\text{-}PSK},C_{3.3}}_{\mathsf{pqWG},n_P,n_S,\mathcal{A}}(\lambda) \leq\ & n_P^2 \cdot n_S^2 \Big(\mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathsf{CCAKEM},\mathcal{R}}(\lambda) \\ & + 7 \cdot \mathsf{Adv}^{\mathsf{prf}}_{\mathsf{KDF},\mathcal{R}}(\lambda) \\ & + 2 \cdot \mathsf{Adv}^{\mathsf{prf}^{\mathsf{swap}}}_{\mathsf{KDF},\mathcal{R}}(\lambda) \Big)\end{aligned}$$

4.7.7. Case 3.4: The Long-term/Ephemeral Subcase

In this subcase we know that (by the definition of the cleanness predicate $\mathsf{clean}_{\mathsf{eCK\text{-}PFS\text{-}PSK}}$ and the subcase preconditions) that the session π_i^s such that the $\mathsf{Test}(i, s)$ session will be queried has an honest contributive keyshare session π_j^t and that $\mathsf{CorruptASK}(i)$ and $\mathsf{CorruptEPK}(j, t)$ queries have not been issued during the execution of the experiment. Note that in our proof we set that the test session has role `init` and the partner session has role `resp`, but the case where the test session has role `resp` and the partner session has role `init` follows analogously. In what follows, we show that in this subcase,

the adversary's probability in winning the key-indistinguishability game is negligible under certain security assumptions.

Game 0 This is a standard eCK-PFS-PSK game with cleanness predicate $\mathsf{clean}_{\mathsf{eCK\text{-}PFS\text{-}PSK}}$ upheld. Thus we have:

$$\mathsf{Adv}^{\mathsf{eCK\text{-}PFS\text{-}PSK}}_{\mathsf{pqWG},n_P,n_S,\mathcal{A}}(\lambda) = \Pr\left[\mathrm{break}_0\right]$$

Game 1 In this game, we guess the index (i, s) of the Test session π_i^s and abort if, during the experiment, a query $\mathsf{Test}(i^*, s^*)$ is issued such that $(i^*, s^*) \neq (i, s)$. Thus:

$$\Pr\left[\mathrm{break}_0\right] \leq n_P \cdot n_S \cdot (\Pr\left[\mathrm{break}_1\right])$$

Game 2 In this game, we guess the index (j, t) of the honest partner session π_j^t (which we know exists by the Case 3 definition) and abort if, during the experiment, a query $\mathsf{Test}(i, s)$ is issued if the contributive keyshare session $\pi_{t^*}^{j^*}$ exists such that $(j^*, t^*) \neq (i, s)$. Thus:

$$\Pr\left[\mathrm{break}_1\right] \leq n_P \cdot n_S \cdot (\Pr\left[\mathrm{break}_2\right])$$

Game 3 In this game we replace the computation of C_8 with uniformly random and independent values $\widetilde{C_8}$. This works almost identical to *Game 5* of **Case 2** and mostly changes labels.

In ***Game 3a*** we replace the value $\hat{r} := \mathsf{KDF}(\sigma_r, r_r)$ passed to CCAKEM.Enc for the computation of `ct1` and `shk1` with a random bitstring $\hat{r}'$.

To show that this replacement is sound, we replace the value of $\hat{r}$ with a uniformly random and independent value $\hat{r}' \xleftarrow{\$} \{0,1\}^{|\mathsf{KDF}|}$ used in the protocol execution of the test session. Specifically, we initialize a $\mathsf{prf}^{\mathsf{swap}}$ challenger and query σ_i, and use the output $\tilde{r}$ from the $\mathsf{prf}^{\mathsf{swap}}$ challenger to replace the computation of $\hat{r}$. By the definition of this case r_r is a uniformly random and independent value, therefore this replacement is sound. If the test bit sampled by the $\mathsf{prf}^{\mathsf{swap}}$ challenger is 0, then $\hat{r} \leftarrow \mathsf{KDF}(\sigma_r, r_r)$ and we are in *Game 2*. If the test bit sampled by the $\mathsf{prf}^{\mathsf{swap}}$ challenger is 1, then $\hat{r} \xleftarrow{\$} \{0,1\}^{|\mathsf{KDF}|}$ is a truly random value and we are in *Game 3a*.

Thus any adversary capable of distinguishing this change can be turned into a successful adversary against the $\mathsf{prf}^{\mathsf{swap}}$ security of KDF, and we find:

$$\Pr\left[\mathrm{break}_2\right] \leq \mathsf{Adv}^{\mathsf{prf}^{\mathsf{swap}}}_{\mathsf{KDF},\mathcal{R}}(\lambda) + \Pr\left[\mathrm{break}_{3a}\right]$$

In ***Game 3b*** we replace the computation of `shk3` by sampling the value uniformly at random from the space of shared secrets of the KEM and ignoring the second output of $\mathsf{CCAKEM.Enc}(\mathtt{spk}_r)$. To show that this is undetectable under the IND-CCA-assumption of the used KEM, we interact with

an IND-CCA challenger in the following way: Note that by *Game 2*, we know at the beginning of the experiment the index of session π_i^s such that $\mathsf{Test}(i, s)$ is issued by the adversary. Similarly, by *Game 1*, we know at the beginning of the experiment the index of the intended partner P_j of the session π_i^s. Thus, we initialize an IND-CCA challenger and use the received public-key $\mathtt{pk}^*$ as long-term public-key of party P_j and give it with all other (honestly generated) public keys to the adversary. Note that by the definition of this case, $\mathcal{A}$ is not able to issue a $\mathsf{CorruptASK}(j)$ query, as we abort if $\pi_i^s.\alpha \leftarrow \mathtt{reject}$ and abort if $\pi_i^s.\alpha \leftarrow \mathtt{accept}$. Thus we will not need to reveal the private key $\mathtt{sk}^*$ of the challenge public-key to $\mathcal{A}$. However we must account for all sessions t such that π_j^t must use the private key for computations. In our version of WireGuard, the long-term private keys are used to compute the following:

- In sessions where P_j acts as the initiator:
  $C_8 \leftarrow \mathsf{KDF}(C_6, \mathsf{CCAKEM.Dec}(\mathtt{ssk}_i, \mathtt{ct3}))$
- In sessions where P_j acts as the responder:
  $C_3, \kappa_3 \leftarrow \mathsf{KDF}(C_2, \mathsf{CCAKEM.Dec}(\mathtt{ssk}_r, \mathtt{ct1}))$

(Note that these are fewer cases than in the original proof because we don't combine static and ephemeral keys directly.) Dealing with the challenger's computation of these values will be done in two ways:

- The encapsulation was created by another honest party. The challenger can then use its own internal knowledge of the encapsulated value to complete the computations.
- The encapsulation was not created by another honest party, but by the adversary and the challenger is therefore unaware of the encapsulated value.

In the second case, the challenger can instead use the decapsulation-oracle provided by the CCA-challenger, specifically querying $\mathsf{CCAKEM.Dec}(\mathtt{ctX})$, (where $\mathtt{ctX}$ is the relevant encapsulation) which will output $\mathtt{shkX}$ using the CCA challenger's internal knowledge of $\mathtt{sk}^*$.

During session i we request a challenge consisting of a ciphertext and a candidate shared secret $(\mathtt{c}^*, \mathtt{k}^*)$ from the IND-CCA challenger and use those values in place of $\mathtt{ct3}$ and $\mathtt{shk3}$. Given the definition of the IND-CCA game, there are two cases:

- If the test bit sampled by the IND-CCA challenger is 0, then $\mathtt{k}^*$ is indeed the shared secret encapsulated in $\mathtt{c}^*$ and we are in *Game 3a*.

- If the test bit sampled by the IND-CCA challenger is 1, then $\mathtt{k}^*$ is not the shared secret encapsulated in $\mathtt{c}^*$ but sampled uniformly at random from the space of shared secrets and we are in *Game 3b*.

Thus, any adversary capable of distinguishing this change can be turned into a successful adversary against the IND-CCA security of the used KEM and we find:

$$\Pr[\text{break}_{3a}] \leq \mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathsf{CCAKEM},\mathcal{R}}(\lambda) + \Pr[\text{break}_{3b}]$$

In ***Game 3c*** we replace the values of $C_8 3$ with uniformly random and independent values $\widetilde{C_8} \xleftarrow{\$} \{0,1\}^{|\mathsf{KDF}|}$ (where $\{0,1\}^{|\mathsf{KDF}|}$ is the output space of the KDF) used in the protocol execution of the test session. Specifically, we initialize a $\mathsf{prf}^{\mathsf{swap}}$ challenger and query $\mathtt{shk3}$, and use the output $\widetilde{C_8}$ from the $\mathsf{prf}^{\mathsf{swap}}$ challenger to replace the computation of C_8. Since by *Game 3b*, $\mathtt{shk3}$ is a uniformly random and independent value, this replacement is sound. If the test bit sampled by the $\mathsf{prf}^{\mathsf{swap}}$ challenger is 0, then $\widetilde{C_8} \leftarrow \mathsf{KDF}(C_7, \mathtt{shk3})$ and we are in *Game 3b*. If the test bit sampled by the $\mathsf{prf}^{\mathsf{swap}}$ challenger is 1, then $\widetilde{C_8} \xleftarrow{\$} \{0,1\}^{|\mathsf{KDF}|}$ and we are in *Game 3c*.

Thus any adversary capable of distinguishing this change can be turned into a successful adversary against the $\mathsf{prf}^{\mathsf{swap}}$ security of KDF, and we find:

$$\Pr[\text{break}_{3b}] \leq \mathsf{Adv}^{\mathsf{prf}^{\mathsf{swap}}}_{\mathsf{KDF},\mathcal{R}}(\lambda) + \Pr[\text{break}_{3c}]$$

Game 4 In this game, we replace the computation of C_9, tmp, κ_9 with uniformly random values $\widetilde{C_9}, \widetilde{tmp}, \widetilde{\kappa_9}$ from the same distribution, in the challenger's execution of the test session π_i^s and its partner session π_j^t. We do so by interacting with a prf challenger in the following way: When it is time to compute $C_9, tmp, \kappa_9 \leftarrow \mathsf{KDF}(\widetilde{C_8}, psk)$ we instead initialize a prf challenger and query it with psk. We note that by *Game 3c* that $\widetilde{C_8}$ is a uniformly random and independent value, and thus this replacement is sound. If the random bit b sampled by the prf challenger is 0, then we are in *Game 3c*. If the random bit b sampled by the prf challenger is 1, then we are in *Game 4*. Any adversary $\mathcal{A}$ capable of distinguishing this change in the experiment can be turned into an algorithm against the prf security of KDF and thus:

$$\Pr[\text{break}_{3c}] \leq \mathsf{Adv}^{\mathsf{prf}}_{\mathsf{KDF},\mathcal{R}}(\lambda) + \Pr[\text{break}_4]$$

Game 5 Similarly to the previous game, we replace the computation of C_{10} with a uniformly random value $\widetilde{C_{10}}$ from the same distribution, in the

challenger's execution of the test session π_i^s and its partner session π_j^t. We do so by interacting with a prf challenger in the following way: When it is time to compute $C_{10} \leftarrow \mathsf{KDF}(C_9, \emptyset)$ we instead initialize a prf challenger and query it with the empty string $\emptyset$. We note that by *Game 4*, $\widetilde{C_9}$ is a uniformly random value independent from the protocol execution, and as such the replacement is sound. If the random bit b sampled by the prf challenger is 0, then we are in *Game 4*. If the random bit b sampled by the prf challenger is 1, then we are in *Game 5*. Any adversary $\mathcal{A}$ capable of distinguishing this change in the experiment can be turned into an algorithm against the prf security of KDF and thus:

$$\Pr[\text{break}_4] \leq \mathsf{Adv}^{\mathsf{prf}}_{\mathsf{KDF},\mathcal{R}}(\lambda) + \Pr[\text{break}_5]$$

Game 6 Similarly to the previous games, we replace the values $tk_i, tk_r \leftarrow \mathsf{KDF}(\widetilde{C_{10}}, \emptyset)$ computed by the challenger in the execution of the test session and its honest contributive keyshare session partner π_j^t with uniformly random values $\widetilde{tk_i}, \widetilde{tk_r}$. We do so by interacting with a prf challenger in the following way: When it is time to compute tk_i, tk_r in the appropriate sessions, we instead initialize a prf challenger and query it with the empty string $\emptyset$. We note that by *Game 5*, $\widetilde{C_{10}}$ is a uniformly random value independent from the protocol execution, and as such the replacement is sound. If the random bit sampled by the prf challenger if 0, then we are in *Game 5*, but otherwise the output of the prf challenger $\widetilde{tk_i}, \widetilde{tk_r}$ is uniformly random and independent and we are in *Game 6*. Any adversary $\mathcal{A}$ capable of distinguishing this change in the experiment can be turned into an algorithm against the prf security of KDF and thus:

$$\Pr[\text{break}_5] \leq \mathsf{Adv}^{\mathsf{prf}}_{\mathsf{KDF},\mathcal{R}}(\lambda) + \Pr[\text{break}_6]$$

Since the response to the $\mathsf{Test}(i, s)$ query issued by the adversary is, in *Game 6*, uniformly random and independent regardless of the test bit b sampled by the challenger, then the adversary's success in winning the key-indistinguishability game is reduced to simply guessing and thus:

$$\Pr[\text{break}_6] = 0$$

$$\begin{aligned}\mathsf{Adv}^{\mathsf{eCK\text{-}PFS\text{-}PSK},C_{3,4}}_{\mathsf{pqWG},n_P,n_S,\mathcal{A}}(\lambda) \leq\ & n_P^2 \cdot n_S^2 \Big(\mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathsf{CCAKEM},\mathcal{R}}(\lambda) \\ & + 3 \cdot \mathsf{Adv}^{\mathsf{prf}}_{\mathsf{KDF},\mathcal{R}}(\lambda) \\ & + 2 \cdot \mathsf{Adv}^{\mathsf{prf}^{\mathsf{swap}}}_{\mathsf{KDF},\mathcal{R}}(\lambda) \Big)\end{aligned}$$

4.7.8. Case 3.5: The Long-terms Subcase

In this subcase we know that (by the definition of $\mathsf{clean}_{\mathsf{eCK\text{-}PFS\text{-}PSK}}$ and the subcase preconditions) that the session π_i^s such that the $\mathsf{Test}(i,s)$ session will be queried has an honest contributive keyshare session π_j^t and that $\mathsf{CorruptASK}(i)$ and $\mathsf{CorruptASK}(j)$ queries have not been issued during the execution of the experiment. In what follows, we show that in this subcase, the adversary's probability in winning the key-indistinguishability game is negligible under certain security assumptions.

Game 0 This is a standard eCK-PFS-PSK game with cleanness predicate $\mathsf{clean}_{\mathsf{eCK\text{-}PFS\text{-}PSK}}$ upheld. Thus we have:

$$\mathsf{Adv}^{\mathsf{eCK\text{-}PFS\text{-}PSK}}_{\mathsf{pqWG},n_P,n_S,\mathcal{A}}(\lambda) = \Pr\left[\mathrm{break}_0\right]$$

Game 1 In this game, we guess the index (i,s) of the Test session π_i^s and abort if, during the experiment, a query $\mathsf{Test}(i^*,s^*)$ is issued such that $(i^*,s^*) \neq (i,s)$. Thus:

$$\Pr\left[\mathrm{break}_0\right] \leq n_P \cdot n_S \cdot (\Pr\left[\mathrm{break}_1\right])$$

Game 2 In this game, we guess the index (j,t) of the honest partner session π_j^t (which we know exists by the Case 3 definition) and abort if, during the experiment, a query $\mathsf{Test}(i,s)$ is issued if the contributive keyshare session $\pi_{t^*}^{j^*}$ exists such that $(j^*,t^*) \neq (i,s)$. Thus:

$$\Pr\left[\mathrm{break}_1\right] \leq n_P \cdot n_S \cdot (\Pr\left[\mathrm{break}_2\right])$$

Game 3 In this game we replace the computation of C_3, κ_3 with uniformly random and independent values $\widetilde{C_3}, \widetilde{\kappa_3}$. This is mostly identical to *Game 5* of case1 and *Game 3* of **Case 3.3**, except that the first subhybrid *Game 3a* differs slightly because we have to assume that σ_i is uncorrupted instead of r_i.

The case is also special because there exists an alternative way to proof it secure: Instead of basing the security on the security of `shk1`, it would also be possible to base it on `shk3`, in which case the proof would resemble those of **Case 2** and **Case 3.4**. For brevity and because there is little to be gained from describing them here in detail as well, we will refrain from doing so.

In ***Game 3a*** we replace the values $\hat{r} := \mathsf{KDF}(\sigma_i, r_i)$ passed to $\mathsf{CCAKEM.Enc}$ for the computation of `ct1` and `shk1` with random bitstrings $\hat{r}'$ in all games of the responder.

We first establish that r_i, while being (potentially) known to the adversary is still fresh in the sense that $\mathsf{KDF}(\sigma_i, r_i)$ has never been evaluated: Since r_i

is a random value, there is a chance that it could be sampled in another session. This probability can be upper-bounded by the total number of sessions divided by the number of possible values, namely $\frac{n_S}{2^\lambda}$ (which when multiplied by the number of sessions results in the famous approximation of the birthday-bound $\frac{n_S^2}{2^\lambda}$).

We do so by interacting with a prf-challenger in the following way: Whenever it is time to compute to compute $\mathsf{KDF}(\sigma_r, X)$ for some value X, we instead query the prf-challenger with X and use the output $\tilde{r}$ from the prf-challenger to replace the computation of $\hat{r}$. By the definition of this case σ_i is a uniformly random and independent value, therefore this replacement is sound.

If the test bit sampled by the prf challenger is 0, then $\hat{r} \leftarrow \mathsf{KDF}(\sigma_i, r_i)$ and we are in *Game 2*. If the test bit sampled by the prf challenger is 1, then $\hat{r} \xleftarrow{\$} \{0,1\}^{|\mathsf{KDF}|}$ is a truly random value. Since we established furthermore that r_i is not used with σ_i in any other session, $\hat{r}$ is furthermore independent of all other $\hat{r}$ in other sessions, therefore we are in *Game 3a*.

Thus any adversary capable of distinguishing this change can be turned into a successful adversary against the prf security of KDF, and we find:

$$\Pr[\text{break}_2] \leq \frac{n_S}{2^\lambda} + \mathsf{Adv}^{\mathsf{prf}}_{\mathsf{KDF},\mathcal{R}}(\lambda) + \Pr[\text{break}_{3a}]$$

Note that this case slightly differs from the previous ones in the same place in that we replace more than just one value with randomness. This is because unlike r_i and r_r, σ_i is used in multiple interactions and thus it becomes necessary to deal with all of them.

In ***Game 3b*** we replace the computation of `shk1` by sampling the value uniformly at random from the space of shared secrets of the KEM and ignoring the second output of $\mathsf{CCAKEM.Enc}(\mathtt{spk}_r)$. To show that this is undetectable under the IND-CCA-assumption of the used KEM, we interact with an IND-CCA challenger in the following way: Note that by *Game 2*, we know at the beginning of the experiment the index of session π_i^s such that $\mathsf{Test}(i,s)$ is issued by the adversary. Similarly, by *Game 1*, we know at the beginning of the experiment the index of the intended partner P_j of the session π_i^s. Thus, we initialize an IND-CCA challenger and use the received public-key $\mathtt{pk}^*$ as long-term public-key of party P_j and give it with all other (honestly generated) public keys to the adversary. Note that by the definition of this case, $\mathcal{A}$ is not able to issue a $\mathsf{CorruptASK}(j)$ query, as we abort if $\pi_i^s.\alpha \leftarrow \mathtt{reject}$ and abort if $\pi_i^s.\alpha \leftarrow \mathtt{accept}$. Thus we will not need to reveal the private key $\mathtt{sk}^*$ of the challenge public-key to $\mathcal{A}$. However we must account for all sessions

t such that π_j^t must use the private key for computations. In our version of WireGuard, the long-term private keys are used to compute the following:

- In sessions where P_j acts as the initiator:
  $C_8 \leftarrow \mathsf{KDF}(C_6, \mathsf{CCAKEM.Dec}(\mathtt{ssk}_i, \mathtt{ct3}))$
- In sessions where P_j acts as the responder:
  $C_3, \kappa_3 \leftarrow \mathsf{KDF}(C_2, \mathsf{CCAKEM.Dec}(\mathtt{ssk}_r, \mathtt{ct1}))$

(Note that these are fewer cases than in the original proof because we don't combine static and ephemeral keys directly.) Dealing with the challenger's computation of these values will be done in two ways:

- The encapsulation was created by another honest party. The challenger can then use its own internal knowledge of the encapsulated value to complete the computations.
- The encapsulation was not created by another honest party, but by the adversary and the challenger is therefore unaware of the encapsulated value.

In the second case, the challenger can instead use the decapsulation-oracle provided by the CCA-challenger, specifically querying $\mathsf{CCAKEM.Dec}(\mathtt{ctX})$, (where $\mathtt{ctX}$ is the relevant encapsulation) which will output $\mathtt{shkX}$ using the CCA challenger's internal knowledge of $\mathtt{sk}^*$.

During session i we request a challenge consisting of a ciphertext and a candidate shared secret $(\mathtt{c}^*, \mathtt{k}^*)$ from the IND-CCA challenger and use those values in place of $\mathtt{ct1}$ and $\mathtt{shk1}$. Given the definition of the IND-CCA game, there are two cases:

- If the test bit sampled by the IND-CCA challenger is 0, then $\mathtt{k}^*$ is indeed the shared secret encapsulated in $\mathtt{c}^*$ and we are in *Game 3a*.
- If the test bit sampled by the IND-CCA challenger is 1, then $\mathtt{k}^*$ is not the shared secret encapsulated in $\mathtt{c}^*$ but sampled uniformly at random from the space of shared secrets and we are in *Game 3b*.

Thus, any adversary capable of distinguishing this change can be turned into a successful adversary against the IND-CCA security of the used KEM and we find:

$$\Pr[\text{break}_{3a}] \leq \mathsf{Adv}^{\text{IND-CCA}}_{\mathsf{CCAKEM},\mathcal{R}}(\lambda) + \Pr[\text{break}_{3b}]$$

In ***Game 3c*** we replace the values of C_3, κ_3 with uniformly random and independent values $\widetilde{C_3}, \widetilde{\kappa_3} \xleftarrow{\$} \{0,1\}^{|\mathsf{KDF}|}$ (where $\{0,1\}^{|\mathsf{KDF}|}$ is the output space of the KDF) used in the protocol execution of the test session. Specifically, we initialize a $\mathsf{prf}^{\mathsf{swap}}$ challenger and query $\mathtt{shk1}$, and use the output $\widetilde{C_3}, \widetilde{\kappa_3}$ from the $\mathsf{prf}^{\mathsf{swap}}$ challenger to replace the computation of C_3, κ_3. Since by *Game 3b*, $\mathtt{shk1}$ is a uniformly random and independent value, this replacement is sound. If the test bit sampled by the $\mathsf{prf}^{\mathsf{swap}}$ challenger is 0, then $\widetilde{C_3}, \widetilde{\kappa_3} \leftarrow \mathsf{KDF}(C_2, \mathtt{shk1})$ and we are in *Game 3b*. If the test bit sampled by the $\mathsf{prf}^{\mathsf{swap}}$ challenger is 1, then $\widetilde{C_3}, \widetilde{\kappa_3} \xleftarrow{\$} \{0,1\}^{|\mathsf{KDF}|}$ and we are in *Game 3c*.

Thus any adversary capable of distinguishing this change can be turned into a successful adversary against the $\mathsf{prf}^{\mathsf{swap}}$ security of KDF, and we find:

$$\Pr[\mathrm{break}_{3b}] \leq \mathsf{Adv}^{\mathsf{prf}^{\mathsf{swap}}}_{\mathsf{KDF},\mathcal{R}}(\lambda) + \Pr[\mathrm{break}_{3c}]$$

Game 4 In this game, we replace the computation of C_6 with a uniformly random value $\widetilde{C_6}$ from the same distribution, in the challenger's execution of the test session π_i^s and its partner session π_j^t. We do so by interacting with a prf challenger in the following way: When it is time to compute $C_6 \leftarrow \mathsf{KDF}(\widetilde{C_4}, \mathtt{ct2})$ we instead initialize a prf challenger and query it with $\mathtt{ct2}$. We note that by *Game 3c* that $\widetilde{C_4}$ is a uniformly random and independent value, and thus this replacement is sound. If the random bit b sampled by the prf challenger is 0, then we are in *Game 4*. If the random bit b sampled by the prf challenger is 1, then we are in *Game 4*. Any adversary $\mathcal{A}$ capable of distinguishing this change in the experiment can be turned into an algorithm against the prf security of KDF and thus:

$$\Pr[\mathrm{break}_{3c}] \leq \mathsf{Adv}^{\mathsf{prf}}_{\mathsf{KDF},\mathcal{R}}(\lambda) + \Pr[\mathrm{break}_4]$$

Game 5 In this game, we replace the computation of C_7 with a uniformly random value $\widetilde{C_7}$ from the same distribution, in the challenger's execution of the test session π_i^s and its partner session π_j^t. We do so by interacting with a prf challenger in the following way: When it is time to compute $C_7 \leftarrow \mathsf{KDF}(\widetilde{C_6}, \mathtt{shk2})$ we instead initialize a prf challenger and query it with $\mathtt{shk2}$. We note that by *Game 4*, $\widetilde{C_6}$ is a uniformly random and independent value, and thus this replacement is sound. If the random bit b sampled by the prf challenger is 0, then $\widetilde{C_7} \leftarrow \mathsf{KDF}(\widetilde{C_6}, \mathtt{shk2})$ and we are in *Game 4*. If the random bit b sampled by the prf challenger is 1, then $\widetilde{C_7} \xleftarrow{\$} \{0,1\}^{|\mathsf{KDF}|}$ and we are in *Game 5*. Any adversary $\mathcal{A}$ capable of distinguishing this change

in the experiment can be turned into an algorithm against the prf security of KDF and thus:

$$\Pr[\text{break}_4] \leq \mathsf{Adv}^{\mathsf{prf}}_{\mathsf{KDF},\mathcal{R}}(\lambda) + \Pr[\text{break}_5]$$

Game 6 Similarly to the previous game, we replace the computation of C_8 with a uniformly random value $\widetilde{C_8}$ from the same distribution, in the challenger's execution of the test session π_i^s and its partner session π_j^t. We do so by interacting with a prf challenger in the following way: When it is time to compute $C_8 \leftarrow \mathsf{KDF}(\widetilde{C_7}, \texttt{shk3})$ we instead initialize a prf challenger and query it with shk3. We note that by *Game 5*, $\widetilde{C_7}$ is a uniformly random and independent value, and thus this replacement is sound. If the random bit b sampled by the prf challenger is 0, then $C_8 \leftarrow \mathsf{KDF}(\widetilde{C_7}, \texttt{shk3})$ and we are in *Game 5*. If the random bit b sampled by the prf challenger is 1, then $\widetilde{C_8} \xleftarrow{\$} \{0,1\}^{|\mathsf{KDF}|}$ and we are in *Game 6*. Any adversary $\mathcal{A}$ capable of distinguishing this change in the experiment can be turned into an algorithm against the prf security of KDF and thus:

$$\Pr[\text{break}_5] \leq \mathsf{Adv}^{\mathsf{prf}}_{\mathsf{KDF},\mathcal{R}}(\lambda) + \Pr[\text{break}_6]$$

Game 7 In this game, we replace the computation of C_9, tmp, κ_9 with uniformly random values $\widetilde{C_9}, \widetilde{tmp}, \widetilde{\kappa_9}$ from the same distribution, in the challenger's execution of the test session π_i^s and its partner session π_j^t. We do so by interacting with a prf challenger in the following way: When it is time to compute $C_9, tmp, \kappa_9 \leftarrow \mathsf{KDF}(\widetilde{C_8}, psk)$ we instead initialize a prf challenger and query it with psk. We note that by *Game 6*, $\widetilde{C_8}$ is a uniformly random and independent value, and thus this replacement is sound. If the random bit b sampled by the prf challenger is 0, then we are in *Game 6*. If the random bit b sampled by the prf challenger is 1, then we are in *Game 7*. Any adversary $\mathcal{A}$ capable of distinguishing this change in the experiment can be turned into an algorithm against the prf security of KDF and thus:

$$\Pr[\text{break}_6] \leq \mathsf{Adv}^{\mathsf{prf}}_{\mathsf{KDF},\mathcal{R}}(\lambda) + \Pr[\text{break}_7]$$

Game 8 Similarly to the previous game, we replace the computation of C_{10} with a uniformly random value $\widetilde{C_{10}}$ from the same distribution, in the challenger's execution of the test session π_i^s and its partner session π_j^t. We do so by interacting with a prf challenger in the following way: When it is time to compute $C_{10} \leftarrow \mathsf{KDF}(C_9, \emptyset)$ we instead initialize a prf challenger and query it with the empty string $\emptyset$. We note that by *Game 8*, $\widetilde{C_9}$ is a uniformly random

value independent from the protocol execution, and as such the replacement is sound If the random bit b sampled by the prf challenger is 0, then we are in *Game 7*. If the random bit b sampled by the prf challenger is 1, then we are in *Game 8*. Any adversary $\mathcal{A}$ capable of distinguishing this change in the experiment can be turned into an algorithm against the prf security of KDF and thus:

$$\Pr[\text{break}_7] \leq \mathsf{Adv}^{\mathsf{prf}}_{\mathsf{KDF},\mathcal{R}}(\lambda) + \Pr[\text{break}_8]$$

Game 9 Similarly to the previous games, we replace the values $tk_i, tk_r \leftarrow \mathsf{KDF}(\widetilde{C_{10}}, \emptyset)$ computed by the challenger in the execution of the test session and its honest contributive keyshare session partner π_j^t with uniformly random values $\widetilde{tk_i}, \widetilde{tk_r}$. We do so by interacting with a prf challenger in the following way: When it is time to compute tk_i, tk_r in the appropriate sessions, we instead initialize a prf challenger and query it with the empty string $\emptyset$. We note that by *Game 8*, $\widetilde{C_{10}}$ is a uniformly random value independent from the protocol execution, and as such the replacement is sound. If the random bit sampled by the prf challenger if 0, then we are in *Game 8*, but otherwise the output of the prf challenger $\widetilde{tk_i}, \widetilde{tk_r}$ is uniformly random and independent and we are in *Game 9*. Any adversary $\mathcal{A}$ capable of distinguishing this change in the experiment can be turned into an algorithm against the prf security of KDF and thus:

$$\Pr[\text{break}_8] \leq \mathsf{Adv}^{\mathsf{prf}}_{\mathsf{KDF},\mathcal{R}}(\lambda) + \Pr[\text{break}_9]$$

Since the response to the $\mathsf{Test}(i, s)$ query issued by the adversary is, in *Game 9*, uniformly random and independent of the test bit b sampled by the challenger, then the adversary's success in winning the key-indistinguishability game is reduced to simply guessing and thus:

$$\Pr[\text{break}_9] = 0$$

$$\begin{aligned}\mathsf{Adv}^{\mathsf{eCK\text{-}PFS\text{-}PSK},C_{3.5}}_{\mathsf{pqWG},n_P,n_S,\mathcal{A}}(\lambda) \leq\ & n_P^2 \cdot n_S^2 \Big(\frac{n_S}{2^\lambda} \\ & + \mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathsf{CCAKEM},\mathscr{R}}(\lambda) \\ & + 7 \cdot \mathsf{Adv}^{\mathsf{prf}}_{\mathsf{KDF},\mathscr{R}}(\lambda) \\ & + \mathsf{Adv}^{\mathsf{prf}^{\mathsf{swap}}}_{\mathsf{KDF},\mathscr{R}}(\lambda) \Big)\end{aligned}$$

5. Post Quantum Noise

This chapter is with the exception of some reordering for practical intents identical to the paper "Post Quantum Noise" [ADH+22a], authored jointly with Yawning Angel, Benjamin Dowling, Andreas Hülsing, and Peter Schwabe, a shortened version of which was published at ACM CCS 2022 [ADH+22b] and which we presented at the Real World Crypto Symposium 2023.

5.1. Introduction

In 2014, Perrin set out to simplify the process of designing, describing, analyzing, and securely implementing secure-channel protocols through the *Noise Protocol Framework* [Per]. The success of this endeavour is illustrated by the long list of users, including WhatsApp, WireGuard, Slack, I2P and the Lightning Network [Moo20]. At the heart of Noise is the idea of using Diffie-Hellman (DH) key exchange [DH76] as the only asymmetric primitive – forward secrecy is achieved through DH with ephemeral keys, authentication of parties is achieved through static DH keys. Perrin informally described this concept of authenticated key agreement without signatures used by Noise as *"Hash all these DHs together to get a final key"* [Per17].

What DH operations are performed and what exactly is "hashed together" is expressed in *handshake patterns*, that Noise specifies in a concise and easy-to-parse language. As one example, consider the "`KN`" pattern:

```
-> s
...
-> e
<- e, ee, se
```

On a high level what this pattern means is that the responder (on the right side of the arrows) is aware of the static public DH key (`-> s`) of the initiator (on the left side of the arrows) before the online phase of the protocol starts (`-> s` is before `...`). In the online phase the initiator first generates an ephemeral DH key pair and sends the ephemeral public key to the responder (`-> e`). The responder then also generates an ephemeral key pair (`e`) and

sends the public key to the initiator (`<- e`). Both parties then combine their respective ephemeral secret keys with the ephemeral public key of the peer to obtain a shared ephemeral-ephemeral DH key (`ee`) and additionally compute the static-ephemeral DH `se` using the initiator's static secret key and the responder's ephemeral public key on the initiator's side and the initiator's static public key and the responder's ephemeral secret key on the responder's side. For a more detailed description of how patterns translate into cryptographic operations and protocols messages, and in particular how public and shared keys are absorbed into protocol state, see the Noise Protocol Framework specification [Per]; for the cryptographic protocol implementing the `KN` pattern, see Figure 5.1.

The `KN` pattern is an example of a *named pattern* in Noise; a subset of these named patterns are the so-called *fundamental patterns*. There exist twelve interactive and three non-interactive fundamental patterns. These patterns exist for every combination of each party having 1) **N**o static public key, 2) a static public key **K**nown to their peer, or 3) a static public key that has to be transmitted (**X**) during the interaction. For the initiator there is furthermore the possibility that 4) he has a static public key that he is willing to send with the **I**nitial message, even if this may reduce anonymity. For each of the resulting 12 cases, Noise defines a fixed pattern, named by the two letter combination derived from concatenating the letters indicating the case for the initiators key and that for the responders key, giving: NN, NK, NX, KN, KK, KX, XN, XK, XX, IN, IK and IX. E.g., NN deals with the case where neither party has a static public key, whereas IK applies to the case where the initiator's key is not initially known to the responder, but the responder's key is known to the initiator upfront, and the initiator is willing to send his public key with the first message.

One interesting feature of secure-channel protocols in Noise is that they do not separate the key-agreement or handshake phase from the data-transmission phase: in fact, Noise allows to send early payload messages together with every handshake message. These early payload messages are encrypted under whatever shared key material has already been established, but they typically do not enjoy the full security properties established by the end of the handshake. This means that we cannot analyze the security of Noise handshakes as standalone, monolithic authenticated-key-agreement protocols in, for example, the CK [CK01], eCK [LLM07], or CK$^+$ [Kra05] model. This issue was addressed by Dowling, Rösler, and Schwenk with the introduction of the fACCE model [DRS20], a multi-stage variant of the ACCE model introduced for the analysis of TLS [JKSS17].

The design decision to rely on Diffie-Hellman as the only asymmetric primitive in Noise leads to elegant protocols offering extensive security and privacy properties; the instantiation of DH with X25519 [Ber06] in Noise also leads to efficient implementations of these protocols in multiple programming languages. However, the strong reliance on DH also comes with a downside: Noise does not have any straight-forward migration path to post-quantum cryptography. Indeed, DH offers the functionality of non-interactive key exchange (NIKE) [FHKP12], and no efficient post-quantum instantiation of this functionality is known today. The most plausible candidate is CSIDH [CLM+18], but unfortunately even at bleeding-edge security levels and with all state-of-the-art optimizations it is about three orders of magnitude slower than X25519 [BBC+21]. Also, the concrete security against quantum attackers is still subject of heavy debate [BS20, Pei20, BLMP19].

5.1.1. Our Contribution.

The closest primitive to DH that *does* have efficient post-quantum instantiations is key encapsulation mechanisms (KEMs). For specific DH-based authenticated key-exchange protocols, KEMs have been used before to replace DH, e.g., in PQWireGuard [HNS+21]; in this chapter we generalize this approach and investigate what a purely KEM-based, post-quantum Noise framework looks like.

While it is straightforward to replace DH by a KEM in some cases, in others it is not, for a multitude of reasons: First, authentication with KEMs can only be done in an interactive challenge-response fashion, whereas it is possible to view any DH public key as an already existing challenge, allowing for non-interactive authentication. Secondly, it is possible to combine arbitrary DH keyshares, which is not the case for KEMs as public keys cannot be combined. This causes issues in the cases where Noise combines two static shares. Thirdly, Noise is extremely flexible and offers a huge number of possible patterns. So far, computational security proofs are given for individual patterns which results in a large number of individual security proofs, and many patterns without computational proofs of security at all, though a number of symbolic analyses of Noise exist [KNB19, GHS+20].

We resolve all of these issues. We provide a recipe to translate a Noise-pattern into a PQNoise pattern that, at the possible cost of additional round-trips, achieves the same confidentiality and authenticity as the original pattern. In some cases, we can do better than applying our generic translation. We provide optimized PQNoise alternatives for all 12 interactive fundamental patterns and for the non-interactive N-pattern (The K- and X-patterns

don't have non-interactive equivalents in PQNoise). Our recipes solve the second issue by noting that approaches like the NAXOS trick provide a way to mix a static secret into the randomness effectively guaranteeing that the result is secret as long as either the randomness or the static secrets are uncorrupted. We introduce *static-ephemeral entropy combination (SEEC)* as an abstraction of these approaches, which is suitable for the security analysis of PQNoise, is met by many existing constructions, and allows the implementer to choose a suitable instantiation for their respective target system.

We give a generic proof of security in the computational model resolving issue three. This is enabled by the introduction of another abstraction termed "hash-object", a formalization of the "Hash all these DHs together to get a final key" idea. A hash-object is a stateful object into which values can be fed and from which keys can be extracted. When using this to analyze PQNoise, we require that the outputs of this object are pseudorandom as long as at least one random input was absorbed into the object before that is unknown to the adversary. We provide a formal definition of this primitive and prove that the way Noise hashes key shares into a hash-chain instantiates it. This abstraction allows us to remove a lot of pattern-specific complexity from the security proofs, which in turn allows us to write them in a generic manner. We conjecture that this approach is fully applicable to all versions of classical Noise, allowing for a more comprehensive computational analysis than what currently exists, though we leave that for future work. We remark here that our proof does in fact not just apply to the specific PQNoise patterns that we specify, but to every PQNoise protocol, including for example hybrid ones (which we don't specify here).

Our security analysis is performed in the fACCE-model [DRS20], that was already used in the analysis of Noise, though we modify the model in a few places. First, there are some cosmetic changes that we believe make both the model and the resulting statements more accessible, such as renaming confusingly named operations. Second, we provide the resulting security statements as a simple table that maps uncorrupted secrets to achieved security goals in a given stage instead of providing a list of named security goals that are also not necessarily independent. This allows us to simplify the freshness conditions significantly.

A proof-of-concept implementation by Angel enabled benchmarks that compare PQNoise to classical Noise [ADH+22b]. The results of these benchmarks clearly demonstrate the practicability of post-quantum key exchanges in a wide variety of settings. We remark here that providing a post-quantum version of Noise essentially provides a solution for all applications that need key exchanges that are requiring neither backwards-compatibility nor crypto-

agility at runtime; naturally the former is not a problem that can be solved generically and the latter is a property whose desirability is getting called increasingly into question as more and more newer protocols do not offer it, which matches community-surveys [Val21].

5.2. PQNoise

In this section we present our design for PQNoise. We start with a description of PQNoise. Afterwards, we introduce SEEC (Static-Ephemeral Entropy Combination), our abstraction of methods that mix a static key into the randomness source to guarantee security in a bad randomness setting. With this we then present our recipe to translate Noise patterns into PQNoise patterns. We conclude with a discussion of the optimized fundamental-patterns for PQNoise.

5.2.1. PQNoise

PQNoise aims to be the post-quantum counterpart to Noise and shares many of its characteristics. One of these is the generic approach of providing a large number of possible patterns whose description is similar to that of Noise patterns. However, given that PQNoise uses KEMs for key exchange, some tokens are different. The single-letter tokens (`s` and `e`) stand for the sending of public keys, just as before. The four tokens (`ee`, `se`, `es` and `ss`) representing combination of DH-key-shares are dropped. In their place PQNoise introduces `ekem` and `skem`, that indicate the sending of a ciphertext that was encapsulated to the ephemeral/static public key of the receiving party and the mixing of the encapsulated secret into the hash-object (our abstraction of the hash-chains used in Noise) similar to the old two-letter tokens.

On a lower abstraction-level PQNoise intentionally works essentially exactly like classical Noise, with the exceptions that we replace the asymmetric primitives and use SEEC for the entropy of all probabilistic algorithms, except the generation of static keys. Noise starts mixing shared keys into its hash chain as soon as they are available, extracts a session key from it and starts encrypting all further messages, except the ephemeral key shares, using an AEAD scheme. We stick to this approach.

Noise and PQNoise maintain effectively two hash-chains (one of which we will later model as a hash-object): The first one, h, is initialized as the hash of a pattern-label. Whenever a value x needs to be added to it, the party in question computes $\mathsf{H}(h, x)$ and replaces h with it. The first thing that

is added to h are unspecified associated data that can be chosen freely by the application. Following that all public keys are added as soon as they are transmitted (if they are **K**nown, they get added at the very start). Furthermore, all AEAD-ciphertexts are added after they are sent/successfully decrypted. In turn h is used directly (without further hashing) as associated data whenever an AEAD-ciphertext is created and is intended to be usable as a unique handshake-hash after the completion of the handshake-phase.

The second hash-chain ck is the one from which the protocol derives its encryption-keys. The key-chain ck is initialized by the hash of the pattern-label as well. Afterwards, whenever both parties establish a shared secret k_i (in classical Noise a Diffie-Hellman shared secret, in PQNoise the key that is encapsulated in a KEM-ciphertext), Noise computes a temporary value (which we will refer to as tmp) as $\mathsf{HMAC\text{-}HASH}(ck, k_i)$ and derives a new value for ck and whatever keys it needs by computing $\mathsf{HMAC\text{-}HASH}(tmp, ctr)$, where ctr is set to 0 for the new value of ck and to 1 for the derived key. There is one exception to this with the last addition of a shared secret, where the two produced values are not used as a new value for ck and a session-key, but instead as the initiator's and responder's session keys for the remaining session. For the purposes of our analysis, we model this as hash-object and refer to Section 5.4.1 for more details.

The actual encryption in PQNoise is done via an AEAD-scheme, where the key is the session-key derived from ck, h is used as associated data and the nonce is a simple counter, that is initially set to zero, increases by 1 with every use and is reset to zero once a new session-key is established. To send an ephemeral key (`e`), the sender creates a new ephemeral keypair pk_e, sk_e using the key-generation-algorithm with the output of SEEC as entropy and adds pk_e to the current payload and h. To send a static key (`s`), the sender adds their static public key to the current payload and h.

Sending of KEM-ciphertexts (`ekem`, `skem`) is where the largest differences between Noise and PQNoise are: Firstly, we differentiate between the ephemeral (EKEM), the initiator's (IKEM) and the responder's (RKEM) KEM. This allows the use of different KEMs in the same protocol in a similar manner to PQWireguard [HNS+21] which can allow for more efficient protocols and enable a "poor man's hybrid encryption", where even a catastrophic break of one scheme preserves confidentiality if there is no additional corruption.) As depicted in Algorithm 10, during Send the sender encapsulates a key $k_{\mathcal{X}}$ to the receiver's public key $pk_{\mathcal{X}}$ using hardened randomness (see Section 5.2.2). If the KEM in question is not ephemeral (for compatibility with Noise) and there is already a shared key k_i (which by the requirements of Noise has to be at least partially derived from EKEM) the resulting ci-

phertext $ct_{\mathcal{X}}$ (together with possible further payload pl that doesn't further affect the KEM-operation, see Appendix B.3) is encrypted with the AEAD-scheme under k_i using the current nonce n and the current handshake-hash h as associated data and the resulting ciphertext is added to the send-buffer. Otherwise $ct_{\mathcal{X}}$ is added directly to the send-buffer. In either case h is updated by hashing the previous value with whatever was added to the send-buffer and $k_{\mathcal{X}}$ is added to the key-chain by calling $ck.\mathsf{in}(k_{\mathcal{X}})$, producing the next secret key k_{i+1}. Lastly the sender sets the nonce n to 0.

The actions by the receiver during Recv mirror those of the sender: After either decrypting or receiving $ct_{\mathcal{X}}$ he adds what he received to h, decapsulates it with his secret key $sk_{\mathcal{X}}$ and inputs the resulting key into the key-chain ck producing k_{i+1} and resets the nonce n to 0.

Algorithm 10: Transmission of KEM-ciphertexts.
Any decryption- or decapsulation-failures lead to an implicit abort.

Function Send:
  …
  $r \leftarrow SEEC.\mathsf{GenRand}(seec\_sk)$
  $ct_{\mathcal{X}}, kk_{\mathcal{X}} := XKEM.\mathsf{encaps}(pk_{\mathcal{X}}, r)$
  if $XKEM \neq EKEM \wedge k_i \neq \perp$**:**
    $c_i := AEAD.enc(k_i, n, h, pl)$
    $h := \mathsf{H}(h, c_i)$
  else:
    $h := \mathsf{H}(h, ct_{\mathcal{X}})$
  $k_{i+1} := ck.\mathsf{in}(kk_{\mathcal{X}})$, $n = 0$
  …
Function Recv:
  …
  if $XKEM \neq EKEM \wedge k_i \neq \perp$**:**
    $\ldots || ct_{\mathcal{X}} := AEAD.dec(k_i, n, h, c_i)$
    $h := \mathsf{H}(h, c_i)$
  else:
    $h := \mathsf{H}(h, ct_{\mathcal{X}})$
  $kk_{\mathcal{X}} := XKEM.\mathsf{decaps}(sk_{\mathcal{X}}, ct_{\mathcal{X}})$
  $k_{i+1} := ck.\mathsf{in}(kk_{\mathcal{X}})$, $n = 0$
  …

We refrain from providing detailed pseudocode for the other operations here as they are essentially identical to classical Noise and refer to Appendix B.3 instead. We note however that we implemented a compiler that transforms any PQNoise-pattern into such detailed pseudocode while also performing some basic soundness-checks (for example that there is no use of keys that are not yet known) on the input and full type-checking on the produced code (though the types are not displayed as part of the LaTeX-output). We provide the pseudocode resulting from the thirteen fundamental PQNoise-patterns (see below) as part of Appendix B.3 and the compiler at `https://fiona.onl/diverses/pqnoise-codegen.tar.bz2`.

To give an illustrative example of how PQNoise and Noise differ we refer to Figure 5.1, which displays the KN-pattern of classical Noise and its PQNoise-counterpart. The main differences can be seen around the use of KEMs: Since KEM-keys cannot be combined, PQNoise requires the sending of additional ciphertexts (ct_e and $ct_\jmath$ instead of just g^b) which also have to be encrypted (c_0) and added to h. That (and the use of SEEC) aside, the protocols are however remarkably similar. Overall, these similarities and differences are representative for the other patterns.

5.2.2. SEEC

Bad random number generators are a real-world issue. And it does not matter for this whether they are intentionally broken by malicious governments [BLN16] or accidentally by well-meaning individuals [DP08]. Hence, this is covered in modern definitions of security for protocols, introducing the corruption of ephemeral secrets as a valid attack.

The Noise-framework itself considers this an issue that should be solved on system level instead of per-protocol and does not include any countermeasures for this case. Nonetheless the KK- and the IK-patterns derive their key among other sources from a static-static Diffie-Hellman exchange. The intention behind this was purely to achieve initiator-authenticity earlier than otherwise possible. However, later analysis [DRS20] came to rely upon it to achieve protection from so called Maximal Exposure (or MEX-) attacks [Kra05], where the adversary can learn the randomness of parties.

Removing this protection from PQNoise would therefore weaken the patterns compared to the security that published analysis promises for their classical counterparts, even if those properties were never promised by the designers of these patterns. As we outlined above, there is no direct replacement for the Static-Static exchange when using KEMs. Nevertheless, similar

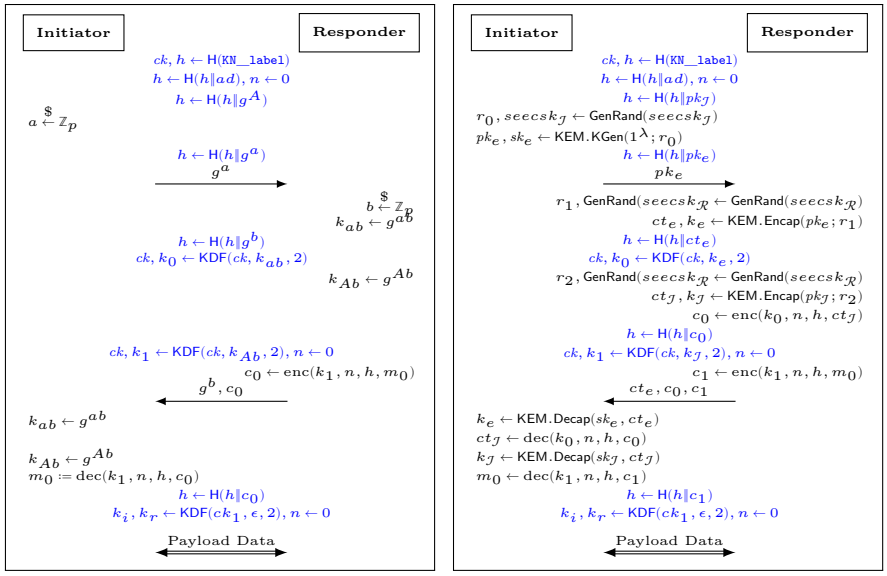

Figure 5.1.: The KN patterns of classical Noise (left) and PQNoise (right). For reasons of space, we use the following conventions here: highlighted actions are performed by both parties as soon as they receive all necessary values to perform the computations in question. If any decryption- or decapsulation-algorithm returns $\perp$, the party in question aborts the protocol.

security properties can be achieved when combining a static secret with the random coins used in the encapsulation algorithm.

The first time something like this was proposed was as part of the NAXOS-protocol [LLM07]. Later Fujioka, Suzuki, Xagawa and Yoneyama [FSXY12] used "twisted PRFs" to achieve a similar result. Later still Akhmetzyanova, Cremers, Garratt, Smyshlyaev and Sullivan [ACG+20] proposed to use hashed signatures of random messages, arguing that the secret keys for signature-schemes often reside in special protected hardware to begin with, making this a practical match. This was then standardized by the IETF as RFC 8937 [CGS+20].

Since the exact choice of such a system should be transparent for all peers, we consider it an implementation detail and refrain from specifying any con-

crete technique. Instead, we introduce the notion of Static-Ephemeral Entropy Combination (SEEC) as an abstraction of all of these and similar approaches and base our analysis on this abstract notion. This allows us to generically analyze PQNoise without forcing implementers to use any specific system. Indeed, SEEC also covers cases where the mixing is done on system level, matching well with the philosophy of Noise while formally describing the requirements to achieve security also under MEX attacks.

Intuitively a SEEC-scheme consists of a pair of algorithms GenKey and $\mathsf{GenRand}$. GenKey is a probabilistic algorithm that returns a long-term key sk. $\mathsf{GenRand}$ then takes sk and some random coins and returns a pseudorandom value r, where r is indistinguishable from a true random value if either sk is uncompromised and the random-coins fresh (but possibly known to the adversary) or if the random-coins are uncompromised. Additionally, we allow but do not require $\mathsf{GenRand}$ to modify sk. The reason is that this is necessary to allow SEEC schemes to implement pre- and post-compromise security. This is a weaker notion than one could strive for, but most existing schemes would not instantiate a stronger notion which would therefore undermine our goal of allowing the implementer to choose freely which one to use.

Definition 47 (SEEC)**.** Formally a SEEC-scheme is a tuple of two algorithms: GenKey and $\mathsf{GenRand}$.

$\mathsf{GenKey}(1^\lambda) \to sk$ takes a security-parameter 1^λ and returns a secret-key sk.

$\mathsf{GenRand}(sk, r) \to (rand, sk)$ is a deterministic algorithm that takes a secret-key sk and a random nonce r and returns a random bit-string $rand$ and a (potentially updated) secret key sk.

As a convention calls to $\mathsf{GenRand}$ that only pass sk as argument shall be considered a shorthand for passing an independently sampled random r.

Definition 48 (SEEC-security)**.** We say that a SEEC-scheme Σ is secure if and only if:

$$\forall \mathcal{A} \in \mathrm{QPT}, \lambda \in \mathbb{N} :$$
$$\Pr\left[\mathsf{Exp}^{SEEC}_{\Sigma,\mathcal{A}}(1^\lambda) = 1\right] - \frac{1}{2} =: \mathsf{Adv}^{SEEC}_{\Sigma,\mathcal{A}}(1^\lambda) \leq \mathrm{negl}(\lambda)$$

Where $\mathsf{Exp}^{SEEC}_{\Sigma,\mathcal{A}}$ is defined as in Experiment 17.

On a high level this definition has similarities to that of a dual-PRF (see Chapter 2.2.5), as it requires the adversary to distinguish the output of a

Experiment 17: $\mathsf{Exp}^{SEEC}_{\Sigma,\mathcal{A}}$, the security experiment for SEEC for a scheme Σ.

$sk := \mathsf{GenKey}(1^\lambda)$
$b \leftarrow_\$ \mathbb{B}$
$randomnesses := []$
$r\_revealed := \emptyset$
$keys\_revealed := 0$
$challenges := \emptyset$
$keys := [sk]$
$i := 0$
Oracle GenRand′**:**
  $r \leftarrow_\$ \mathbb{B}^\lambda$
  $randomnesses[i] := r$
  $i := i + 1$
  $ret, sk := \mathsf{GenRand}(sk, r)$
  $keys[i] := sk$
  return ret
Oracle getKeys**:**
  $keys\_revealed := 1$
  return $keys$
Oracle getCoins *(j)***:**
  abort if $(j \notin \{0, \ldots, i-1\})$
  $r\_revealed \cup = \{j\}$
  return $r[j]$
Oracle Challenge**:**
  $r \leftarrow_\$ \mathbb{B}^\lambda$
  $R_0, sk := \mathsf{GenRand}(sk, r)$
  $R_1 \leftarrow_\$ \mathbb{B}^\lambda$
  $randomnesses[i] := r$
  $challenges \cup = \{i\}$
  $i := i + 1$
  return R_b
$b' := \mathcal{A}^{\mathsf{GenRand}',\mathsf{getKey},\mathsf{getCoins},\mathsf{Challenge}}(1^\lambda)$
if $keys\_revealed \wedge (r\_revealed \cap challenges \neq \emptyset)$**:**
  return 0
return $b = b'$

function that receives one random and one known input from randomness. A closer look reveals several differences however: Firstly, we allow the algorithm to update the long-term secret to enable the use of SEEC schemes that evolve their keys. Secondly, we do not allow the adversary to choose either argument himself, as we currently only aim to give protection from predictable, but not from non-uniform randomness. Lastly and most importantly we allow free mixing of real queries and challenge queries that may or may not return real outputs. This is a desirable property for the use in protocols, where a party may use both fully exposed and partially unexposed secrets in different sessions. By allowing free mixing of the queries, it becomes possible to have fully exposed outputs, while those with partially unknown inputs can be replaced by random values.

We specify for PQNoise that a SEEC-scheme may be used for both the generation of the ephemeral keypairs and all randomness used in key-encapsulation with the remark that not using it breaks security in settings where ephemeral randomness can be compromised; since that may sometimes be considered acceptable, we provide a full analysis of both cases.

We provide PRP-SEEC in Appendix B.2 as a simple and efficient instantiation of SEEC, note however that since the specific scheme in question is fully hidden from the peer and our proof is fully generic, using a different approach such as the one presented in RFC 8937 is viable and may sometimes be preferable (see [ACG+20] for a discussion).

5.2.3. Translating Patterns

Given the general description of PQNoise, it remains to be shown how we move from a Noise pattern to a PQNoise pattern, generically. While the translation of most steps is straightforward, there are two non-trivial cases that have to be handled with care. The first one is any instance of a Static-Ephemeral or Ephemeral-Static exchange, where the sending-party is the owner of the static key. In the DH case they immediately prove the identity of the sender (assuming the static key is uncompromised), establishing authenticity right away. This does not work with KEMs, as the owner of the static public key cannot combine their secret key with their peer's ephemeral public key. The obvious workaround of having their peer create and send the ciphertext (as in the case where the sending party is owner of the ephemeral key) is not equivalent as it does not yet confirm the authenticity of the owner of the static key. Instead, the owner has to send an additional key-confirmation message (i.e., if this was the last message, another AEAD ciphertext from the

static key owner using a key derived from the encapsulated value is necessary). In effect this adds up to one roundtrip.

The second one is replacing a Static-Static exchange, as there is no direct equivalent for it. However, using and extending the technique from the previous paragraph we can create a workaround that achieves similar security. In Noise the Static-Static exchange establishes authenticity for both parties and confidentiality (assuming uncompromised static keys). Sending encapsulations for both static KEMs would almost establish the same properties as long as we secure the coins used by the encapsulating party using SEEC with a static secret. The resulting shared secrets are unknown to the adversary as long as the static keys are uncompromised. Afterwards, a key confirmation is necessary by the initial sender. Given that this adds a full roundtrip and that `ss` is usually used to get an early shared secret before a roundtrip, it may sometimes be more reasonable to drop this combination entirely, using `se` and `es` for authenticity.

Everything else can usually stay as it is, with the exception that the transmission of the responder's ephemeral public key can be dropped entirely. (This is under the assumption that the initiator sends an ephemeral public key before the responder does; otherwise, the initiator's ephemeral public key gets dropped. As it would delay the arrival at forward secrecy, we see little reason to deviate from that convention and are unaware of any proposals to use such patterns.)

This gives us the following recipe to translate patterns in a manner that we conjecture to preserve the security ("conjecture" because while we prove the security of the PQNoise patterns, we lack a generic proof of classical Noise to compare the results to):

Ephemeral-Ephemeral exchanges (`ee`) can be directly replaced by sending a ciphertext for the ephemeral KEM (`ekem`).

Ephemeral-Static exchanges (`es`) sent by the initiator and Static-Ephemeral exchanges (`se`) sent by the responder can be directly replaced by sending a ciphertext for the receiving party's KEM (`skem`).

Ephemeral-Static exchanges (`es`) sent by the responder and Static-Ephemeral exchanges (`se`) sent by the initiator are more complicated: When the initiator in Noise sends `se`, we replace this by the initiator finishing his current turn, following to which the responder sends `skem`, the initiator replies with a key-confirmation that may also contain the remaining operations given in the line of the original pattern. The same approach is used for `es` sent by the responder, with reversed roles.

Static-Static exchanges (`ss`) where the initiator is the original sender, are replaced as follows. The initiator sends `skem`, computing her coins using SEEC with a static secret and ends her turn. The responder responds with `skem`, the coins also obtained using SEEC with a static secret, and ends his term. Lastly the initiator has to send another message for key confirmation. If the responder is the original sender, roles are reversed.

After removing duplicate actions (usually multiple uses of `skem`), we conjecture the resulting pattern to achieve the same confidentiality, authenticity, integrity, anonymity and deniability as the original one; While we don't further analyze the later three goals, the lack of a generic analysis of classical noise (which is out of scope for this work) prevents us from proving the first two. Our generic analysis does however show that PQNoise matches (or exceeds) the conjectured / proven security [DRS20] for those original Noise-patterns, that have been analyzed in the fACCE-model. This may come at the disadvantage of only achieving that security at a later point in the interaction, due to the additional roundtrips.

One property that cannot be preserved by this translation-approach is that the ephemeral key-share of a party is used with both shares of their peer; Because of this the peer could be certain that the derived keys belong to the same ephemeral key using DH. This is no longer the case with KEMs since there is in general no way to derive useful information about the used ephemeral entropy from the ciphertext and reusing entropy may even introduce vulnerabilities. In our formal analysis this does not cause problems for confidentiality and authenticity. However, protocol designers who rely upon the dual-use of the ephemeral keys in Noise for other purposes need to be aware of this.

5.2.4. Fundamental Patterns

With the above recipe we can convert any Noise pattern into a PQNoise-pattern. The result of this transformation is, however, not always optimal in terms of roundtrips. For this reason, we hand-picked a PQNoise-pattern to match each of the twelve fundamental classical Noise-patterns ($\{I, N, K, X\} \times \{N, K, X\}$). These PQNoise patterns are designed to not only achieve at least the same amount of confidentiality and authenticity, but to also do that as efficiently as possible. (The equivalent security follows from the use of the same KEMs, for which our proofs show that each KEM will introduce some degree of security independent of the order of their use, but possibly at different protocol-stages.)

All of them ended up having a direct equivalent in classical Noise when looking beyond its own fundamental patterns from which they result as part of the generic translation: The IK and the KK patterns are equivalent to IKnoss and KKnoss [PC18]. All patterns that involve (non-early) transmitted keys are equivalent to the deferred patterns where the transmitted key sees deferred use (every "X" becomes "X1"), for example IX is equivalent to IX1 and XX is equivalent to X1X1. All other patterns are equivalent to their namesakes.

While some of these patterns require more roundtrips than their classical counterparts and may achieve certain degrees of security at a slightly later point, they all eventually end up achieving the same degree of security that was conjectured or proven [DRS20] for their classical counterparts.

While we initially hand-picked the fundamental PQNoise-patterns and eventually found them equivalent to certain classical patterns, we also identified the following process to arrive at them, that only considers the scenario in which the keys are used:

- When a party knows a public key that belongs to their peer, that party's next message will always include a ciphertext for that public key, if none has been sent already.
- $\mathcal{I}$ sends an ephemeral public key in the first message.
- If $\mathcal{R}$ has a static public key that is not known to $\mathcal{I}$, it is sent in the second message.
- If $\mathcal{I}$ has a static public key that is not known to $\mathcal{R}$ and does not require anonymity (`I*`-pattern), that public key is sent as part of the first message.
- If $\mathcal{I}$ has a static public key that is not known to $\mathcal{R}$ and does require anonymity, it is sent as part of the third message.
- Within a message `ekem`, if present, always precedes `skem` which, if present, always precedes all public keys and the payload.

We include it here as it may be useful for the design of extended versions of PQNoise that may for example make use of signatures.

Noise also provides three non-interactive patterns ($\{N, K, X\}$). The authenticated K- and X-patterns cannot be translated into non-interactive versions of PQNoise, as the initiator cannot prove his identity in a non-interactive way using only KEMs. The unauthenticated N-pattern can however be translated trivially and essentially results in the standard KEM/DEM-construction.

We note that our analysis applies to the N-pattern as well and therefore include it in the list of the thirteen fundamental PQNoise patterns.

We depict the interactive ones of these in Figure 5.2 and provide more detailed descriptions in Appendix B.3

```
pqNN:
-> e
<- ekem

pqNK:
<- s
...
-> skem, e
<- ekem

pqNX:
-> e
<- ekem, s
-> skem

pqKN:
-> s
...
-> e
<- ekem, skem

pqKK:
-> s
<- s
...
-> skem, e
<- ekem, skem

pqKX:
-> s
...
-> e
<- ekem, skem, s
-> skem

pqXN:
-> e
<- ekem
-> s
<- skem

pqXK:
<- s
...
-> skem, e
<- ekem
-> s
<- skem

pqXX:
-> e
<- ekem, s
-> skem, s
<- skem

pqIN:
-> e, s
<- ekem, skem

pqIK:
<- s
...
-> skem, e, s
<- ekem, skem

pqIX:
-> e, s
<- ekem, skem, s
-> skem
```

Figure 5.2.: The interactive fundamental PQNoise patterns.

5.3. Overview of the Flexible ACCE Framework

We analyze the security of PQNoise in the flexible authenticated and confidential channel establishment (fACCE) framework [DRS20] which was developed for the analysis of Noise. Here we give a high-level overview of the cryptographic primitive fACCE, and define fACCE security, highlighting areas that we have modified for our specific setting of post-quantum channel-establishment protocols.

The main divergence between the original fACCE model and our version is how we structure and represent *freshness conditions*. These allow the protocol analyser to determine in which settings an attack is valid, i.e. after what set of compromises or adversary actions is the adversary considered to win the game. This is largely determined by a definition of when a protocol is supposed to achieve a certain security goal. In the original fACCE model, this was represented as a series of freshness *counters*, which captured confidentiality or authentication under certain types of attacks e.g. $\texttt{au}^{\rho}$ defines when the party with role ρ authenticates itself, and thus when it is considered a non-trivial attack that the adversary can inject or modify messages from the ρ-party. This resulted (in their full model) in ten counters, each capturing a specific type of attack and compromise paradigm.

We instead represent all combinations of secrets (long-term and ephemeral) for each session as rows in a *security table* (ST), with *authentication* and *confidentiality* columns. For each combination of secrets, we indicate in which stage(s) authentication and confidentiality hold if the adversary *has not* compromised those secrets. This results in a simpler, more intuitive representation as it focuses on the natural question: *Under any given compromise strategy, when does the protocol achieve (if at all) confidentiality and authenticity?* instead of requiring the reader (or protocol designer) to understand and interpret cryptographic history (e.g., the `eck` counter describes an adversary that can compromise either session's ephemeral randomness). We also use copies of ST (which we denote the *freshness table* or FT) as a tool within our formalism, serving to simplify our freshness conditions: each session begins with a full FT, and whenever an adversary compromises a particular type of secret, the rows with that secret are removed from FT. An adversary that attempts to break the security of stages that are not associated with some combination of secrets are considered invalid as the result of trivial attacks. In addition to this, we make a small number of mostly aesthetic changes:

- We rename Enc and Dec as Send and Recv. We note that this better matches their semantics, as Send and Recv also transmit channel-establishment material, and potentially do not perform encryption and decryption at all, depending on the protocol.
- We require that each Send operation increments the stage counter of the channel. The original fACCE model only incremented the stage counter when new (and increased) security properties are reached. This change ties the stage of the channel to its flow in the channel communication. As a result, we modify the definition of fACCE protocols to no longer output the stage counter ς when sending or receiving messages, as it is sufficient to count the messages between communicating parties.

We now turn to describing the fACCE primitive and security framework on a high-level and give some additional insight into the changes made to the freshness conditions. The full model can be found in Appendix B.1.

fACCE Primitive Description.

On a high-level, fACCE is a cryptographic protocol that both establishes a secure channel and provides authenticated and confidential communication between two parties. Eschewing a modular approach, channel establishment and payload transmission are handled by the same algorithms – where Send sends channel establishment information and (potentially encrypted) payload data, and Recv receives. These functions may also update the internal state of the sessions. For a complete definition we refer to Appendix B.1, but we privde a shortened version here:

Formally a flexible ACCE protocol fACCE is a tuple of four algorithms $\mathsf{KGen}, \mathsf{Init}, \mathsf{Send}, \mathsf{Recv}$ associated with a long-term secret key space $\mathcal{LSK}$, a long-term public key space $\mathcal{LPK}$, an ephemeral secret key space $\mathcal{ESK}$ an ephemeral public key space $\mathcal{EPK}$, and a state space $\mathcal{ST}$. The definition of fACCE algorithms are as follows:

$\mathsf{KGen} \rightarrow_{\$} (pk, sk)$ generates long-term keys where $pk \in \mathcal{LPK}, sk \in \mathcal{LSK}$. Note that this captures both long-term asymmetric key pairs, as well as potential long-term symmetric secrets (which we consider a part of sk).

$\mathsf{Init}(sk, ppk, \rho, ad) \rightarrow_{\$} st$ initializes a session to begin communication, where sk (optionally) are the initiator's long-term secret keys, ppk (optionally) is the long-term public key of the intended session partner, $\rho \in \{\mathtt{i}, \mathtt{r}\}$ is the session's role (i.e., initiator or responder), ad is data associated with

this session, and $sk \in \mathcal{LSK} \cup \{\perp\}$, $ppk \in \mathcal{LPK} \cup \{\perp\}$, $ad \in \{0,1\}^*$, $st \in \mathcal{ST}$.

$\mathsf{Send}(sk, st, m) \rightarrow_\$ (st', c)$ continues the protocol execution in a session and takes a message m to output a new state st', and messages c[1], where $sk \in \mathcal{LSK} \cup \{\perp\}$, $st, st' \in \mathcal{ST}, m, c \in \{0,1\}^*$. Note that Send may generate additional ephemeral key pairs $(epk, esk) \in \mathcal{EPK} \times \mathcal{ESK}$[2].

$\mathsf{Recv}(sk, st, c) \rightarrow_\$ (st', m)$ processes the protocol execution in a session triggered by c and outputs new state st', and message m, where $sk \in \mathcal{LSK} \cup \{\perp\}$, $st \in \mathcal{ST}$, $st' \in \mathcal{ST} \cup \{\perp\}$, $m, c \in \{0,1\}^*$. If $st' = \perp$ is output, then this denotes a rejection of this ciphertext.

We assume messages sent in fACCE are sent in a ping-pong fashion, i.e., the initiator sends a message to the responder, who replies to the initiator, and so on. Multiple messages in a single flow are thus extensions of a single message. Each message monotonically increases the stage of the protocol, i.e., the first message sent from initiator to responder is stage one, the first message sent from responder to initiator is stage two, etc. This differs from the original fACCE, which only increments stages when achieving new security properties.

We define the correctness of an fACCE protocol in Appendix B.1, Definition 55. Intuitively an fACCE protocol is correct if messages sent from the established channel were equally accepted by their partner.

Execution Environment.

Here we describe (on a high-level) the execution environment for our fACCE security experiment. We consider a set of n_P parties each (potentially) maintaining a long-term key pair $\{(sk_1, pk_1), \ldots, (sk_{n_P}, pk_{n_P})\}$, $(sk_i, pk_i) \in \mathcal{LSK} \times \mathcal{LPK}$. Each party can participate in up to n_S sessions, with each session potentially lasting n_T stages. Each session samples per-session randomness *rand* used throughout the protocol execution. We denote both the set of variables that are specific for a session s of party i as well as the identifier of this session as π_i^s. Further details on the session state can be found in Appendix B.1.

Honest partnering is defined over the transcript sent between two sessions. Intuitively, a session has an honest partner if all ciphertexts the honest partner

[1] Note that messages here may consist of channel establishment data (such as keying material), encrypted payload data, or even plaintext payload data. In what follows, we refer to these generically as "ciphertexts", even when sending plaintext data.

[2] In the security experiment, these are stored within state st

received were sent by the session (without modification) and vice versa, and at least one party received a ciphertext at least once. The full definition of honest partner can be found in Definition 56 in Appendix B.1.

The fACCE model can capture authentication and confidentiality under various compromise paradigms, similar to the *levels* of authentication and confidentiality encoded by the original fACCE's various counters. We also highlight that this approach aligns with the typical structures of proofs of fACCE protocols – when one of the right-hand columns is not ∞, this represents a case distinction in the proof. This proof structure is common in the analysis of authenticated key exchange protocols, especially those in the extended-Canetti-Krawczyk (eCK) model [LLM07], such as the proofs of WireGuard [DP18] and PQWireGuard [HNS+21].

To facilitate the security game, the challenger maintains for each session π_i^s a set $\mathbb{S}_{\pi_i^s}$ that contains labels of all secrets that each session (and its honest partner) maintains – the long-term secret values sk_i, sk_j (both asymmetric and symmetric), all ephemeral secret values sampled during the n_T stages of the protocol execution $esk_s^1, esk_t^1 \ldots, esk_s^{n_T}, esk_t^{n_T}$ and the state maintained during the protocol executions at each stage $st_s^1, st_t^1 \ldots, st_s^{n_T}, st_t^{n_T}$. Thus $\mathbb{S}_{\pi_i^s}$ $= (sk_i, sk_j, esk_s^1, esk_t^1, \ldots, esk_s^{n_T}, esk_s^{n_T}, st_s^1, st_t^1 \ldots, st_s^{n_T}, st_t^{n_T})$.

Each session in an fACCE experiment is associated with a four column freshness table FT (a copy of the original ST), with each element of the powerset of $\mathbb{S}_{\pi_i^s}$ (labels for each secret for itself and its honest partner) contained in the left column, and stage counters / tuples in the *Confidentiality*, *Authenticity of Initiator*, and *Authenticity of Responder* columns. The intuition here is that the table declares at which stages confidentiality and authenticity (for each role) are achieved under the assumption that the associated combinations of secrets have *not* been compromised by an attacker.

Consider the NK Noise Pattern ST displayed in Table 5.1. In the table we denote the ephemeral Diffie-Hellman secret value that the initiator samples as $e_{\mathcal{I}}$ and the responder samples as $e_{\mathcal{R}}$, and the long-term Diffie-Hellman secret value that the responder maintains (B) as $s_{\mathcal{R}}$. If (at least) the long-term key of the responder $s_{\mathcal{R}}$ and the ephemeral key of the initiator $e_{\mathcal{I}}$ remain uncompromised the NK Pattern achieves responder authentication in stage $\varsigma = 2$ and does not achieve initiator authentication. If $e_{\mathcal{R}}$, $e_{\mathcal{I}}$ remain uncompromised, NK achieves confidentiality in stage $\varsigma = 2$, and if $s_{\mathcal{R}}$ and $e_{\mathcal{I}}$ remain uncompromised then NK achieves confidentiality of messages in stage $\varsigma = 1$. The intuition on an attacker's winning condition is that if the adversary breaks security in any stages associated with a particular combination of secrets that have not been compromised, the adversary wins.

Table 5.1.: The NK Noise Pattern (`<-s \\... \\->e, es \\<-e, ee`) and associated fACCE security table.

Secrets	Conf	Auth - `i`	Auth - `r`
$s_{\mathcal{R}}$	∞	∞	∞
$s_{\mathcal{R}}$, $e_{\mathcal{I}}$	1	∞	2
$s_{\mathcal{R}}$, $e_{\mathcal{R}}$	∞	∞	∞
$e_{\mathcal{I}}$, $e_{\mathcal{R}}$	2	∞	∞
$s_{\mathcal{R}}$, $e_{\mathcal{I}}$, $e_{\mathcal{R}}$	1	∞	2

Adversarial Model.

In order to model active attacks in our environment, the security experiment provides the OInit, OSend, ORecv oracles to an adversary $\mathcal{A}$, who can use them to control communication among sessions, together with the oracles $\mathsf{OCorrupt}$, $\mathsf{OReveal}$ and $\mathsf{ORevealRandomness}$.

Following the direction of the original fACCE work, we treat the authentication and confidentiality properties similarly to the original AEAD notion of Rogaway [Rog02]: the game maintains a win flag (to indicate whether the adversary broke authenticity or integrity of ciphertexts) and changes encryption behaviour based on randomly sampled challenge bits (to model indistinguishability of ciphertexts). In order to win the security game, adversary $\mathcal{A}$ either has to trigger $\mathsf{win} \leftarrow 1$ or output the correct challenge bit $\pi_i^s.b_\varsigma$ of a specific session stage ς at the end of the game.

In addition, the challenger maintains a set of freshness flags $\pi_i^s.fr_\varsigma$ for each stage ς of each session π_i^s. When $\mathcal{A}$ makes a query to $\mathsf{OCorrupt}$, $\mathsf{OReveal}$ or $\mathsf{ORevealRandomness}$, then $\mathcal{C}$ deletes all rows in the freshness table FT that contain the secret revealed to $\mathcal{A}$. All stages for all sessions that are not an element of the right-hand columns are now considered un-fresh, and the corresponding freshness flags are set to 0. When $\mathcal{A}$ terminates and outputs a session π_i^s and a stage counter ς such that the freshness flag associated with $\pi_i^s.\varsigma$ is 0, then $\mathcal{C}$ simply outputs a random bit b^* instead of $\pi_i^s.b_\varsigma = b'$.

We describe the function of each oracle below. The details on excluding trivial attacks as the result of these oracles can be found in Appendix B.1.

$\mathsf{OInit}(i, pk_j, \rho, ad)$ initializes a new session π_i^s (if not yet initialized) of party i to be partnered with party j, invoking

fACCE.$\mathsf{Init}(sk_i, pk_j, \rho, ad) \rightarrow_{[\pi_i^s.rand]} \pi_i^s.st$ using randomness $\pi_i^s.rand$ and returning the index of the session s.

$\mathsf{OSend}(i, s, m_0, m_1)$ triggers the encryption of the message m_b where $b = \pi_i^s.b_\varsigma$ by invoking $\mathsf{Send}(sk_i, \pi_i^s.st, m_b) \rightarrow_{[\pi_i^s.rand]} (st', c)$ for an initialized π_i^s if $|m_0| = |m_1|$. Note that c contains both the explicit ciphertext encryption of the message m_b and any channel establishment messages that are sent in this stage. Finally, c is appended to $\pi_i^s.T_s$.

$\mathsf{ORecv}(i, s, c)$ triggers invocation of $\mathsf{Recv}(sk_i, \pi_i^s.st, c) \rightarrow_{[\pi_i^s.rand]} (st', m)$ for an initialized π_i^s and returns (m, ς) only if π_i^s has no honest partner, and returns ς if an honest partner exists. If an honest partner exists, and the session is currently fresh, then outputting the plaintext message m would leak the challenge bit, so we must prevent this leakage. The adversary breaks authentication (and thereby $\mathsf{win} \leftarrow 1$ is set) if the received ciphertext was not sent by a session of the intended partner but was successfully received (i.e., there exists no honest partner and the output state is $st' \neq \perp$), and $\mathcal{A}$ has not issued queries that trivially break authentication in this stage. Finally, c is appended to $\pi_i^s.T_r$ if decryption succeeds.

$\mathsf{ORevealRandomness}(i, s) \rightarrow rand$ outputs the ephemeral randomness $rand$ sampled by session π_i^s. The freshness table FT and freshness flags are updated by the challenger.

$\mathsf{OCorrupt}(i) \rightarrow sk_i$ outputs the long-term secret key sk_i of party i and updates the freshness table FT and freshness flags.

$\mathsf{OReveal}(i, s) \rightarrow \pi_i^s.st$ outputs the current session state $\pi_i^s.st$ and updates the freshness table FT and freshness flags.

Finally, we formalise the security of an fACCE primitive in Appendix B.1. A flexible ACCE protocol fACCE is post-quantum secure if it is correct and $\mathsf{Adv}_{\mathcal{Q}}^{\mathrm{fACCE}}$ is negligible for all quantum algorithms $\mathcal{Q}$ running in polynomial-time.

5.4. Analysis

In this section we present our security analysis of PQNoise. To begin, we model Noise's use of key-derivation-functions as a "hash-object". This allows us to separate the analysis of Noise into the analysis of the hash-object, which

focuses on the local key derivation activities of a user, and the analysis of the key exchange executed between users.

We note that besides the key-derivation-chain ("key-chain") Noise also computes a second hash-chain h to create a handshake-hash; the modelling and analysis in the following section do not apply to that chain.

5.4.1. Hash-Object

Noise has a somewhat convoluted key derivation process as it derives fresh symmetric keys every time it computes a new shared key. Towards this end, Noise makes use of a key-chain into which all shared secrets are absorbed and from which all session-keys are extracted. This chain effectively is a PRF chain in which a previous chaining value is used as key, and any new input is used as input. The output is split into an output and a new chaining value. In an analysis this can be treated as a series of independent pseudorandom function calls. However, the proofs that result from this approach tend to have a long sequence of game hops applying the dual-PRF assumption to replace PRF outputs by random values based on the chaining value or the input being pseudorandom. These are shared by many proof-steps and distract from the core part of the different proofs. Because of this, we introduce an abstraction that allows us to treat such chains as a single object with new security properties that allow to prove security of protocols like (PQ)Noise. We call the new object a hash-object, provide a definition of pseudorandomness for such objects, and prove that the construction of a hash-object used in Noise achieves this property.

Noise usually creates multiple outputs whenever it inputs a new value into its hash-chain, the first of which is usually used as a form of a state that we model as the state s of our hash-object. At the end of the handshake-phase Noise uses the first result directly as output and forgoes the creation of a new state. To model this, we introduced a function $\mathsf{finalize}$ that mostly behaves like the regular input-function, except that it does not return a new state.

Definition 49 (Hash-Object)**.** Formally a hash-object is a tuple of three deterministic algorithms: create, input, $\mathsf{finalize}$, and an integer-constant n.

$\mathsf{create}(1^\lambda) \to s$ takes a security-parameter λ and returns a state s.

$\mathsf{input}(s, m) \to s', h$ takes a state s and message $m \in \mathbb{B}^*$ and returns a new state s' and a list $h \in (\mathbb{B}^\lambda)^n$ of hashes of length n .

$\mathsf{finalize}(s, m) \to h$ works like input, except that it does not return a state.

For convenience sake we will use class-style notation, with an implicit update of the first argument (i.e. $h := s.\mathsf{input}(m)$ instead of $s, h := \mathsf{input}(s, m)$).

Definition 50 (Pseudorandom Hash-Object)**.** We say that a hash-object HO is a pseudorandom hash-object if and only if $\forall \mathcal{A} \in \mathrm{QPT}, \lambda \in \mathbb{N}$:
$\left(\begin{array}{l} \Pr\left[\mathsf{Exp}^{PRHO}_{\mathsf{HO},\mathcal{A},1}(1^\lambda) = 1\right] \\ -\Pr\left[\mathsf{Exp}^{PRHO}_{\mathsf{HO},\mathcal{A},0}(1^\lambda) = 1\right] \end{array} \right) =: \mathsf{Adv}^{PRHO}_{\mathsf{HO},\mathcal{A}}(1^\lambda) \leq \mathrm{negl}(\lambda)$ where $\mathsf{Exp}^{PRHO}_{\mathsf{HO},\mathcal{A}}$ is defined as in Experiment 18.

The core idea behind this definition is that the adversary receives oracle-access to an arbitrary number of hash-objects into which he can feed whatever values he likes. At any point in time, he can request to add the random secret r that is sampled once at the start of the game to any oracle by invoking Rand. From that point onwards all the outputs from the randomized hash-object will either be true random values or real, depending on the challenge-bit. Everything else in this definition is just there to prevent trivial attacks: *history* keeps track of the exact queries performed on each hash-object. *queries* is a dictionary that saves the set of queries that were previously performed on hash-objects with a given history to prevent running both In and Fin on objects in the same state, as the later would reveal the resulting state of the former. *cache* is a dictionary that is used to ensure that two hash-objects with the same history always return the same results even if they have been randomized and return truly random values.

With this we define the Noise Hash Object as depicted in Algorithm 11. This is more or less a direct recreation of how Noise defines HKDF, except that it distinguishes the case where the first argument is then used as state from the case where no state is maintained, and everything is returned as output.

Theorem 12. *A Noise Hash Object NHO is a secure pseudo-random Hash-Object if* $\mathsf{HMAC-HASH}$ *is a dual-prf with:*

$$\mathsf{Adv}^{PRHO}_{NHO,\mathcal{A},q_i}(1^\lambda) \leq \left(\begin{array}{l} \mathsf{Adv}^{\mathit{CollRes}}_{\mathsf{HMAC-HASH},\mathcal{A}'}(1^\lambda) + \\ \mathsf{Adv}^{\mathit{PRF\text{-}SWAP}}_{\mathsf{HMAC-HASH},\mathcal{A}'}(1^\lambda) + \\ (2 \cdot q) \cdot \mathsf{Adv}^{\mathit{PRF}}_{\mathsf{HMAC-HASH},\mathcal{A}'}(1^\lambda) \end{array} \right)$$

where q refers to the total number of oracle-queries.

(Intuitively the collision-resistance of HMAC-HASH implies that only identical histories result in equal states and HMAC-HASH being a dual-PRF (see Chapter 2.2.5) ensures that once r has been added to a chain, its first state becomes pseudorandom which is retained upon subsequent calls.)

Experiment 18: $\mathsf{Exp}^{PRHO}_{\mathsf{HO},\mathcal{A},b}$, the pseudo-randomness experiment for a hash-object HO.

```
r ←$ B^λ
hashes := [], history := [], j := 0
randomized := [0, ..., 0]
finalized := [0, ..., 0]
queries := ∅
cache := dict()
return A^{Create,In,Fin,Rand}(1^λ)

Oracle Create:
    i, j := j, j + 1
    hashes[i] := create(1^λ)
    history[i] := []
    return i

Oracle Rand(i, finalize):
    randomized[i] := 1
    if finalize:
        return Fin(i, r)
    else:
        return In(i, r)

Oracle In(i, m):
    abort_if(finalized[i]
      ∨ (history[i] || ("Fin", m)) ∈
      queries)
    history[i].append("In", m)
    queries ∪= history[i]
    if randomized[i]:
        if cache[history[i]] ≠ ⊥:
            return cache[history[i]]
        h_0 := hashes[i].input(m)
        h_1 ←$ (B^λ)^n
        cache[history[i]] := h_b
        return h_b
    else:
        return hashes[i].input(m)

Oracle Fin(i, m):
    abort_if(finalized[i]
      ∨ (history[i] || ("In", m)) ∈
      queries)
    finalized[i] := 1
    history[i].append("Fin", m)
    queries ∪= history[i]
    if randomized [i]:
        if cache[history[i]] ≠ ⊥:
            return cache[history[i]]
        h_0 := hashes[i].finalize(m)
        h_1 ←$ (B^λ)^n
        cache[history[i]] := h_b
        return h_b
    else:
        return hashes[i].finalize(m)
```

Algorithm 11: Noise-compatible instantiation for the pseudo-random hash-object.

```
1  Function create(1^λ):
2      return ""
3  Function finalize(state, m):
4      (h_0, [h_1. ... , h_n]) := input(state, m)
5      return [h_0. ... , h_{n-1}])
6  Function input(state, m):
7      tmp := HMAC-HASH(state, m)
8      last := ""
9      for i ∈ {0, ... , n}:
10         h_i := HMAC-HASH(tmp, last||byte(i))
11         last := h_i
12     return h_0, [h_1. ... , h_n]
```

Proof. We use game-hopping to show the claim. Let q_{X} be the total number of calls to the X-oracle and $\Pr[\text{diff}_Y]$ to the difference in probability that an adversary $\mathcal{A}$ outputs 1 between game Y and the previous game.

Let ***Game 0*** refer to the regular PRHO-game with the challenge-bit b being set to 0.

In ***Game 1*** we abort if there are ever two evaluations of $\mathsf{HMAC-HASH}$ that produce the same output. To see that this modification is undetectable we initialize a collision-resistance-challenger and use its $\mathsf{HMAC-HASH}$ for this protocol. If there are ever two different histories that produce the same output, then there have to be two different inputs to $\mathsf{HMAC-HASH}$ that share an output by the pigeon-hole principle. We can therefore use that collision to win the collision-resistance game and find:

$$\Pr[\text{diff}_1] \leq \mathsf{Adv}^{\mathsf{CollRes}}_{\mathsf{HMAC-HASH},\mathcal{A}'}(1^\lambda)$$

In ***Game 2*** we replace the values of *tmp*, that are computed during the oracle-invocations of Rand with random (but consistent in case of equal values for *state*) values. We remark that these values are always computed in the same way, irrespective of whether it is part of a call to input or finalize. To show that this replacement is sound we initialize a PRF-SWAP-Challenger for $\mathsf{HMAC-HASH}$ and query it with *state* instead of computing *tmp* directly. Since the message-part of the original invocation is a truly random bitstring

this replacement is sound. If the PRF-SWAP-Challenger's challenge-bit is zero, then it samples r randomly and computes tmp as $\mathsf{HMAC-HASH}(state, r)$ and we are in *Game 1*. Otherwise, the values for tmp are sampled at random, and we are in *Game 2*. Thus, we find:

$$\Pr[\mathrm{diff}_2] \leq \mathsf{Adv}^{\text{PRF-SWAP}}_{\mathsf{HMAC-HASH},\mathcal{A}'}(1^\lambda)$$

In ***Game 3*** we replace the results of all invocations of $\mathsf{HMAC-HASH}$ that use one of the tmp that is random by *Game 2* as key with random and independent values. Given that we replaced q_{Rand} different instances of tmp that may or may not all have different values we will now use q_{Rand} sub-games to replace all the values that are derived from them with randomness. Let *Game 2.0* = *Game 2* and tmp_i be the i'th tmp that is created in the game and has a distinct value.

Then in ***Game 2.***i we replace all outputs of $\mathsf{HMAC-HASH}$ when called with tmp_i as key, with random values if tmp_i is unique. If there is more than one instance of In, Fin, or any combination of them during whose execution we replace values, we replace corresponding values with the same random value (this works because there are no collisions by *Game 1*). To show that this replacement is sound we initialize a PRF-Challenger for $\mathsf{HMAC-HASH}$ and query it with m whenever we would compute $\mathsf{HMAC-HASH}(tmp, m)$ in *Game 2.*$(i-1)$. By *Game 2.*$(i-1)$, tmp_i is a truly random bitstring and all messages within an individual oracle-invocation are different due to the appended counter. Moreover, the game does not allow querying input and $\mathsf{finalize}$-oracles with the same history and there are no colliding histories that produce the same tmp_i by *Game 1*. Hence, this replacement is sound. If the PRF-Challenger's challenge-bit is zero, then it returns $\mathsf{HMAC-HASH}(tmp_i, m)$ for each input m and we are in *Game 2.*$(i-1)$. Otherwise, the outputs are sampled at random, and we are in *Game 2.*i. Thus, we find:

$$\Pr[\mathrm{diff}_{2.i}] \leq \mathsf{Adv}^{\text{PRF}}_{\mathsf{HMAC-HASH},\mathcal{A}'}(1^\lambda)$$

Since there are at most q_{Rand} different sub-games, we can conclude by setting *Game 3* = *Game 2.*q_{Rand} and summarizing the losses of all sub-games that:

$$\Pr[\mathrm{diff}_3] \leq q_{\mathsf{Rand}} \cdot \mathsf{Adv}^{\text{PRF}}_{\mathsf{HMAC-HASH},\mathcal{A}'}(1^\lambda)$$

In ***Game 4*** we replace the outputs of all invocations of In and Fin where the hash-object has been randomized with randomness. To do so we will define J sub-games, where $J \leq 2(q_{\mathsf{In}} + q_{\mathsf{Fin}})$ is the total number of all such invocations in all chains. We set *Game 3.0.1* := *Game 3*, iterate j from 1 to J and note that *Game 3.J.1* = *Game 4*.

In ***Game 3.j.0*** (following *Game 3.*$(j-1)$*.1*) we replace the value of *tmp* that is used during the computation of $\mathsf{input}/\mathsf{finalize}$ in the j'th invocation of In and Fin (=: tmp_j) with a random value, except for maintaining consistency between identical invocations of $\mathsf{HMAC}-\mathsf{HASH}$. To show that this replacement is sound we initialize a PRF-Challenger for $\mathsf{HMAC}-\mathsf{HASH}$ and replace the invocation of $\mathsf{HMAC}-\mathsf{HASH}$ that produces tmp_j by an invocation of the PRF-oracle and replace all instances where *tmp* is produced by the same history as tmp_j with the output of that invocation. Since $state[i]$ is random and independent at the latest by the previous game, this replacement is valid. If the PRF-Challenger's challenge-bit is zero, then the oracle returns $\mathsf{HMAC}-\mathsf{HASH}(k, state[i])$ with a random key r and we are in *Game 3.*$(j-1)$*.1*. Otherwise, the returned values are random we are in *Game 3.j.0*. Thus, we find:

$$\Pr\left[\mathrm{diff}_{3.j.1}\right] \leq \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{HMAC}-\mathsf{HASH},\mathcal{A}'}\left(1^{\lambda}\right)$$

In ***Game 3.j.1*** we replace the return-values and resulting states of all invocations of input and $\mathsf{finalize}$ that use the value of tmp_j (as defined in the previous sub-game) by random values, such that identical invocations of $\mathsf{HMAC}-\mathsf{HASH}$ do however use the same values. To show that this replacement is sound we initialize a PRF-Challenger for $\mathsf{HMAC}-\mathsf{HASH}$ and replace all calls using tmp_j as key with invocations of the oracle provided by the challenger. Since tmp_j is random and independent by the previous game, this replacement is valid. If the PRF-Challenger's challenge-bit is zero, then it answers all queries for a value m̆ by computing $\mathsf{HMAC}-\mathsf{HASH}(k, m)$ with a random key r and we are in *Game 3.j*-1.1. Otherwise, the returned values are random we are in *Game 3.j.0*. Thus, we find:

$$\Pr\left[\mathrm{diff}_{3.j.1}\right] \leq \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{HMAC}-\mathsf{HASH},\mathcal{A}'}\left(1^{\lambda}\right)$$

At this point all randomized outputs are truly random as they would be if the challenge-bit b was 1, meaning that by noting that $q = q_{\mathsf{In}}+q_{\mathsf{Fin}}+q_{\mathsf{Rand}}$ and summarizing all losses, we can then find the claim stated in Theorem 12. □

5.4.2. PQNoise

At this point we can now start the analysis of PQNoise itself. We consider PQNoise with and without the use of SEEC. The reason for analyzing both is that Noise has traditionally considered bad RNGs a problem of the operating system which combined with the fact that the use of SEEC is (if there is no corruption) unobservable from the outside, suggests that the Noise-project

may refuse to specify the use of SEEC and leave it as an implementation-detail.

Let Π be a PQNoise-protocol and Π' be the same protocol without the use of SEEC. Let $\#I$, $\#R$ and $\#E$ refer to the stage of Π/Π' during which the KEM-ciphertexts for the initiator's/ responder's/ ephemeral public keys are sent and ∞ if they are not sent. Let n_P be the number of parties participating in a protocol, n_S be the maximum number of sessions a party participates in, and n_K the total number of session-keys that a party uses. We are using standard-definitions for AEAD, PRFs, PRF-SWAPs, and KEMs (see Chapter 2.2). On top of that we use the definition of pseudo-random hash-object (PRHO) from above.

Intuitively the following four theorems can be summarized like this: In PQ-Noise, authenticity for a party $\mathcal{P}$ is established once it sends a valid reply to a message that was encrypted with uncorrupted randomness under $\mathcal{P}$'s uncorrupted public key. This is because $\mathcal{P}$'s peer $\mathcal{U}$ is by the definition and requirements of this case an honest peer whose KEM-ciphertext is fresh and can only be decrypted by $\mathcal{P}$. $\mathcal{P}$'s response contains an AEAD-ciphertext whose key is derived from the shared secret that only $\mathcal{P}$ and $\mathcal{U}$ have access too. As AEAD-ciphertexts cannot be forged without the key, and $\mathcal{U}$ knows that the reply was not created by her, she can, by the corruption-setting, conclude that she is talking to $\mathcal{P}$.

Theorem 13. Π *achieves initiator-authenticity in stage* $\#I + 1$ *if the initiator's static key and either of the responder's keys have not been corrupted with:*

$$\mathsf{Adv}^{fACCE.auth_{\mathcal{I}}}_{\Pi,\mathcal{A}}(1^\lambda) \leq \mathsf{Adv}^{\mathsf{CollRes}}_{\mathsf{H},\mathcal{A}'}(1^\lambda) + \tfrac{4(n_P \cdot n_S)^2}{2^\lambda} + n_P^2 \cdot n_S \cdot (\mathsf{Adv}^{SEEC}_{\Sigma,\mathcal{A}'}(1^\lambda) + IKEM.\delta + \mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{IKEM,\mathcal{A}',n_S}(1^\lambda) + \mathsf{Adv}^{\mathsf{PRHO}}_{H,\mathcal{A}'}(1^\lambda) + n_K \cdot \mathsf{Adv}^{\mathsf{auth}}_{AEAD,\mathcal{A}'}(1^\lambda))$$

Theorem 14. Π' *achieves initiator-authenticity in stage* $\#I + 1$ *if the initiator's static key and the responder's ephemeral key have not been corrupted with:*

$$\mathsf{Adv}^{fACCE.auth_{\mathcal{I}}}_{\Pi,\mathcal{A}}(1^\lambda) \leq \mathsf{Adv}^{\mathsf{CollRes}}_{\mathsf{H},\mathcal{A}'}(1^\lambda) + \tfrac{4(n_P \cdot n_S)^2}{2^\lambda} + n_P^2 \cdot n_S \cdot (IKEM.\delta + \mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{IKEM,\mathcal{A}',n_S}(1^\lambda) + \mathsf{Adv}^{\mathsf{PRHO}}_{H,\mathcal{A}'}(1^\lambda) + n_K \cdot \mathsf{Adv}^{\mathsf{auth}}_{AEAD,\mathcal{A}'}(1^\lambda))$$

Theorem 15. Π *achieves responder-authenticity in stage* $\#R + 1$ *if the responder's static key and either of the initiator's keys have not been corrupted with:*

$$\mathsf{Adv}^{fACCE.auth_{\mathcal{R}}}_{\Pi,\mathcal{A}}(1^\lambda) \leq \mathsf{Adv}^{\mathsf{CollRes}}_{\mathsf{H},\mathcal{A}'}(1^\lambda) + \tfrac{4(n_P \cdot n_S)^2}{2^\lambda} + n_P^2 \cdot n_S \cdot (\mathsf{Adv}^{SEEC}_{\Sigma,\mathcal{A}'}(1^\lambda) + RKEM.\delta + \mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{RKEM,\mathcal{A}',n_S}(1^\lambda) + \mathsf{Adv}^{\mathsf{PRHO}}_{H,\mathcal{A}'}(1^\lambda) + n_K \cdot \mathsf{Adv}^{\mathsf{auth}}_{AEAD,\mathcal{A}'}(1^\lambda))$$

Theorem 16. Π' *achieves responder-authenticity in stage* $\#R+1$ *if the responder's static key and the initiator's ephemeral key have not been corrupted with:*
$\mathsf{Adv}^{fACCE.auth_{\mathcal{R}}}_{\Pi,\mathcal{A}}(1^\lambda) \leq \mathsf{Adv}^{\mathsf{CollRes}}_{\mathsf{H},\mathcal{A}'}(1^\lambda) + \frac{4(n_P \cdot n_S)^2}{2^\lambda} + n_P^2 \cdot n_S \cdot (RKEM.\delta + \mathsf{Adv}^{\mathit{IND\text{-}CCA}}_{RKEM,\mathcal{A}',n_S}(1^\lambda) + \mathsf{Adv}^{\mathit{PRHO}}_{H,\mathcal{A}'}(1^\lambda) + n_K \cdot \mathsf{Adv}^{\mathit{auth}}_{AEAD,\mathcal{A}'}(1^\lambda))$

Intuitively the security is a consequence of the ciphertext for the initiator's/ responder's static public key being generated with good randomness for an uncorrupted key. The resulting shared secret is then fed into the hash-object whose outputs can be treated as random from then on. Eventually the adversary would therefore have to break the authenticity of the AEAD-scheme in order to create a message as the key is essentially random at that point. The relatively low tightness is a consequence of having to guess the attacked session and parties.

Proof. We show the theorems by contradiction, assuming that there is an adversary that can cause $win \leftarrow 1$ to be set. For this we use game hopping: Let $\mathcal{B}$ be the honest party and $\mathcal{C}$ be the potentially impersonated party ($\mathcal{B}$ would be the Responder and $\mathcal{C}$ would be the Initiator in Theorem 13, and in Theorem 15 it would be the other way around). Let XKEM refer to the KEM instance used with $\mathcal{C}$'s static key and $\Pr[\text{break}_X]$ be the adversarial advantage in winning *Game X*.

Game 0 refers to the original fACCE-game.

In ***Game 1*** we abort if there is ever a hash-collision for hash-chain H. To show that this replacement is sound we initialize a collision-resistance-challenger for H and output any collision to it. We find:

$$\Pr[\text{break}_0] \leq \Pr[\text{break}_1] + \mathsf{Adv}^{\mathsf{CollRes}}_{\mathsf{H},\mathcal{A}'}(1^\lambda)$$

In ***Game 2*** we abort if there is ever a collision of the ephemeral entropy. Even though we allow for corrupted randomness, we assume that it is still properly distributed. Thus, the probability of a collision is given by a birthday-bound for two parties per session, n_S sessions per party (not necessarily with an honest partner) and n_P parties, giving us:

$$\Pr[\text{break}_1] \leq \Pr[\text{break}_2] + \frac{(2 \cdot n_P \cdot n_S)^2}{2^\lambda}$$

In ***Game 3*** we guess $\mathcal{B}$ and $\mathcal{C}$ as well as the session in which $\mathcal{B}$ is targeted by adversary $\mathcal{A}$ and `win = 1` is set. We abort if the guess is wrong and find:

$$\Pr[\text{break}_2] \leq n_P^2 \cdot n_S \cdot \Pr[\text{break}_3]$$

In ***Game 4*** we replace the randomness used for key-encapsulation with $\mathcal{C}$'s public key with true randomness if SEEC is used and don't do anything otherwise. To show that this replacement is sound, we initialize a SEEC-challenger for the used SEEC-scheme Σ and replace all of $\mathcal{B}$'s computations of GenRand in other sessions with invocations of GenRand′, use the getKey- and getRandomness-oracles to answer any corruption-queries by $\mathcal{A}$ and replace the GenRand-call for the encapsulation with an invocation of the Challenge-oracle. If the challenge-bit b is zero, then the encapsulation-randomness is computed via GenRand and we are in *Game 3*. Otherwise, it is a true random value, and we are in *Game 4*. Since $\mathcal{A}$ is not allowed to corrupt both $\mathcal{B}$'s static key and $\mathcal{B}$'s ephemeral key in the target-session, this reduction follows the rules of the SEEC-experiment and we find:

$$\Pr\left[\text{break}_3\right] \leq \Pr\left[\text{break}_4\right] + \mathsf{Adv}^{SEEC}_{\Sigma,\mathcal{A}'}\left(1^\lambda\right)$$

Alternatively, if SEEC is not used we don't change the game and note that we are only considering cases where the randomness is uncorrupted in the first place, giving us:

$$\Pr\left[\text{break}_3\right] = \Pr\left[\text{break}_4\right]$$

In ***Game 5*** we replace the key $k_{\mathcal{C}}$ encapsulated in the ciphertext $c_{\mathcal{C}}$ under $\mathcal{C}$'s public key with a uniformly random key. To show that this replacement is sound we initialize an ICC-CCA-challenger for XKEM and replace $\mathcal{C}$'s static public key with the challenge public key. Whenever $\mathcal{C}$ needs to perform a decapsulation we use the decapsulation-oracle which will by definition occur at most n_S times. This substitution is valid since the encapsulation of the challenge-ciphertext uses true randomness by *Game 4* and the static secret key is, by the definition of this case, not corrupted and only used for decapsulations that can easily be replaced with oracle-calls. If the challenge-bit is zero, then all operations are still performed as before, and we are in *Game 4*. Otherwise, the key has been replaced with an independent random value and we are in *Game 5*. Thus, we find:

$$\Pr\left[\text{break}_4\right] \leq \Pr\left[\text{break}_5\right] + \mathsf{Adv}^{\mathsf{ICC\text{-}CCA}}_{XKEM,\mathcal{A}',n_S}\left(1^\lambda\right)$$

In ***Game 6*** we replace all outputs of the (implicit) hash-object after inputting $k_{\mathcal{C}}$, the previously replaced key, with random values. To show that this replacement is sound we initialize a PRHO-challenger for HO and replace $\mathcal{C}$'s direct use of HO with the oracles in the following way: Whenever $\mathcal{C}$ starts a session and would normally initialize a hash-object, she will instead call Create and use the returned identifier i_{HO} for all oracle invocations in that

session. Whenever $\mathcal{C}$ would normally use the input/finalize functions of HO she will instead invoke the In/Fin oracle, with one exception: When she would normally input $k_{\mathcal{C}}$, she will instead invoke Rand. This substitution is valid since $k_{\mathcal{C}}$ is an independent random value by *Game 5*. If the challenge bit b of the PRHO game is 0, this is a purely conceptual change and we are in *Game 5*. Otherwise, all outputs after inputting $k_{\mathcal{C}}$ get replaced with independent random values and we are in *Game 6*. Thus:

$$\Pr\left[\text{break}_5\right] \leq \Pr\left[\text{break}_6\right] + \mathsf{Adv}^{PRHO}_{\mathsf{HO},\mathcal{A}'}\left(1^\lambda\right)$$

(We remark here that the replaced keys may be repeated in later sessions in which they are consistently replaced as well.)

In ***Game 7*** we guess the session key $\hat{k}$ that is used last in the message that $\mathcal{A}$ forges. If the message involved KEM-operations, there may be multiple keys involved in creating it, the key relevant here is the one used at the very end. In the event that $\mathcal{A}$ fully impersonated $\mathcal{C}$ from the start, this will usually have to be the next message, but in the event of honest partnered sessions $\mathcal{A}$ could inject a forged message at any later point in time. There are by definition n_K session keys in total, which forms a lower bound of $\frac{1}{n_K}$ for guessing correctly and we find:

$$\Pr\left[\text{break}_6\right] \leq n_K \cdot \Pr\left[\text{break}_7\right]$$

In ***Game 8*** we abort after receiving a message that is supposed to be at least partially encrypted under $\hat{k}$ but was not sent by $\mathcal{C}$. To show that this replacement is sound we initialize an auth-challenger for the AEAD-scheme and replace all encryptions that $\mathcal{B}$ and $\mathcal{C}$ would perform with $\hat{k}$ as key with calls to the encryption-oracle and replace all decryptions by using the knowledge of the plaintext of the honestly generated ciphertexts. This replacement is valid since $\hat{k}$ is a fresh random value since *Game 6* and the nonces of all ciphertexts under $\hat{k}$ that $\mathcal{B}$ created have distinct nonces that have a lower value than the nonce for which $\hat{c}$ has to be valid. If $\mathcal{C}$ has been an honest peer until this point, the same holds true for all of her ciphertexts, except for the most recent one, which has the correct nonce. We remark here that this ciphertext will use h as associated data and that since there are no collisions by *Game 1*, everything before the ciphertext is either unmodified relative to the message $\mathcal{C}$ generated (and h therefore the same between the ciphertexts) or it is modified and the associated data therefore different, making the new ciphertext a valid forgery. If h is unmodified on the other hand, then the ciphertext itself has to be different, as the message would otherwise be identical to the one $\mathcal{C}$ sent. Since we are looking at the last ciphertext that is part of the message

and since there is already an established shared secret, there is no further information that could be changed. As a consequence of all this, if $\mathcal{A}$ in any way modifies the message $\hat{c}$ is a fresh ciphertext or has fresh associated data that can be forwarded to the auth-challenger. Thus, if $\hat{c}$ is a valid ciphertext and not part of a non-honest message we win the authenticity game for the AEAD-scheme. Otherwise, the game either remains honest or would abort anyways both preventing $\mathcal{A}$ from winning, since neither sets *win* to 1. We therefore find:

$$\Pr\left[\text{break}_7\right] \leq \mathsf{Adv}^{\mathsf{auth}}_{AEAD,\mathcal{A}'}\left(1^\lambda\right)$$

By summarizing these losses and applying Lemma 1 we find the adversarial advantage stated in Theorems 13-16. □

Intuitively the next six theorems state that confidentiality is achieved once a KEM-ciphertext for an uncorrupted keypair is sent. As the stage in which these are sent depends on the pattern, the actual stage at which messages are confidential depends on it too and on the corruption in question. To get the first confidential stage, one has to pick the lowest stage of the applicable results given below (if the conditions for a result are unmet because of unacceptable corruption or non-use of the associated KEM, that stage is ∞). The first four of these theorems deal with uncorrupted static keys.

Theorem 17. Π *achieves confidentiality in stage #I if either of the responder's and the static key of the initiator is uncorrupted and the responder is an honest party with:*

$$\begin{aligned}\mathsf{Adv}^{fACCE.conf_\jmath}_{\Pi',\mathcal{A}}\left(1^\lambda\right) \leq & \frac{4(n_P \cdot n_S)^2}{2^\lambda} + \mathsf{Adv}^{\mathsf{CollRes}}_{\mathsf{H},\mathcal{A}'}\left(1^\lambda\right) \\ & + n_P^2 \cdot n_S^2 \cdot \left(\begin{array}{l} \mathsf{Adv}^{SEEC}_{\Sigma,\mathcal{A}'}\left(1^\lambda\right) \\ +IKEM.\delta + \mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{IKEM,\mathcal{A}',n_S}\left(1^\lambda\right) \\ +\mathsf{Adv}^{\mathsf{PRHO}}_{H,\mathcal{A}'}\left(1^\lambda\right) \\ +(n_S-1) \cdot \mathsf{Adv}^{\mathsf{auth}}_{AEAD,\mathcal{A}'}\left(1^\lambda\right) \\ +n_K \cdot \mathsf{Adv}^{\mathsf{IND\$\text{-}CPA}}_{AEAD,\mathcal{A}'}\left(1^\lambda\right) \end{array} \right),\end{aligned}$$

where n_K is the total number of session-keys used in a session and n'_K refers to the total number of session keys that have been used before the key encapsulated in IKEM has been put into the hash-object.

Theorem 18. Π' *achieves confidentiality in stage #I if the responder's ephemeral key and the static key of the initiator are uncorrupted, and the*

responder is an honest party with:

$$\mathsf{Adv}^{fACCE.conf_{\mathcal{I}}}_{\Pi',\mathcal{A}}(1^\lambda) \leq \frac{4(n_P \cdot n_S)^2}{2^\lambda} + \mathsf{Adv}^{\mathsf{CollRes}}_{\mathsf{H},\mathcal{A}'}(1^\lambda) + n_P^2 \cdot n_S^2 \cdot \begin{pmatrix} IKEM.\delta + \mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{IKEM,\mathcal{A}',n_S}(1^\lambda) \\ +\mathsf{Adv}^{\mathsf{PRHO}}_{H,\mathcal{A}'}(1^\lambda) \\ +(n_S-1) \cdot \mathsf{Adv}^{\mathsf{auth}}_{AEAD,\mathcal{A}'}(1^\lambda) \\ +n_K \cdot \mathsf{Adv}^{\mathsf{IND\$\text{-}CPA}}_{AEAD,\mathcal{A}'}(1^\lambda) \end{pmatrix},$$

where n_K is the total number of session-keys used in a session and n'_K refers to the total number of session keys that have been used before the key encapsulated in IKEM has been put into the hash-object.

Theorem 19. Π *achieves confidentiality in stage $\#R$ if either of the initiator's keys and the static key of the responder is uncorrupted and the initiator is an honest party with:*

$$\mathsf{Adv}^{fACCE.conf_{\mathcal{R}}}_{\Pi',\mathcal{A}}(1^\lambda) \leq \frac{4(n_P \cdot n_S)^2}{2^\lambda} + \mathsf{Adv}^{\mathsf{CollRes}}_{\mathsf{H},\mathcal{A}'}(1^\lambda) + n_P^2 \cdot n_S^2 \cdot \begin{pmatrix} \mathsf{Adv}^{SEEC}_{\Sigma,\mathcal{A}'}(1^\lambda) \\ +RKEM.\delta + \mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{RKEM,\mathcal{A}',n_S}(1^\lambda) \\ +\mathsf{Adv}^{\mathsf{PRHO}}_{H,\mathcal{A}'}(1^\lambda) \\ +(n_S-1) \cdot \mathsf{Adv}^{\mathsf{auth}}_{AEAD,\mathcal{A}'}(1^\lambda) \\ +n_K \cdot \mathsf{Adv}^{\mathsf{IND\$\text{-}CPA}}_{AEAD,\mathcal{A}'}(1^\lambda) \end{pmatrix},$$

where n_K is the total number of session-keys used in a session and n'_K refers to the total number of session keys that have been used before the key encapsulated in RKEM has been put into the hash-object.

Theorem 20. Π' *achieves confidentiality in stage $\#R$ if the initiator's ephemeral key and the static key of the responder are uncorrupted, and the initiator is an honest party with:*

$$\mathsf{Adv}^{fACCE.conf_{\mathcal{R}}}_{\Pi',\mathcal{A}}(1^\lambda) \leq \frac{4(n_P \cdot n_S)^2}{2^\lambda} + \mathsf{Adv}^{\mathsf{CollRes}}_{\mathsf{H},\mathcal{A}'}(1^\lambda) + n_P^2 \cdot n_S^2 \cdot \begin{pmatrix} RKEM.\delta + \mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{RKEM,\mathcal{A}',n_S}(1^\lambda) \\ +\frac{4(n_P \cdot n_S)^2}{2^\lambda} \\ +\mathsf{Adv}^{\mathsf{PRHO}}_{H,\mathcal{A}'}(1^\lambda) \\ +(n_S-1) \cdot \mathsf{Adv}^{\mathsf{auth}}_{AEAD,\mathcal{A}'}(1^\lambda) \\ +n_K \cdot \mathsf{Adv}^{\mathsf{IND\$\text{-}CPA}}_{AEAD,\mathcal{A}'}(1^\lambda) \end{pmatrix},$$

where n_K is the total number of session-keys used in a session and n'_K refers to the total number of session keys that have been used before the key encapsulated in RKEM has been put into the hash-object.

The proofs are largely similar to the previous ones, the main-difference being that instead of relying on the unforgeability they now need to rely on the confidentiality (IND$-CPA) of the AEAD-scheme. The increased loss in tightness is a result of having to guess the peer's session and the attacked AEAD-key on top of what the previous proofs had to guess.

Proof. We use game hopping to show the statements. Let $\mathcal{B}$ be the assumed honest party and $\mathcal{C}$ be the party who owns the assumed uncorrupted key. (For Theorems 17 and 18, $\mathcal{B}$ would be the Responder and $\mathcal{C}$ would be the Initiator, and, for Theorem 19 and 20, it would be the other way around.) Let XKEM refer to the static KEM of $\mathcal{C}$.

Game 0 refers to the original fACCE-game.

In ***Game 1*** we abort if there is ever a hash-collision for hash-chain H. To show that this replacement is sound we initialize a collision-resistance-challenger for H and output any collision to it. We find:

$$\Pr[\text{break}_0] \leq \Pr[\text{break}_1] + \mathsf{Adv}^{\mathsf{CollRes}}_{\mathsf{H},\mathcal{A}'}(1^\lambda)$$

In ***Game 2*** we abort if there is ever a collision of the ephemeral entropy. The probability of this is given by a birthday-bound and we find:

$$\Pr[\text{break}_1] \leq \Pr[\text{break}_2] + \frac{(2 \cdot n_P \cdot n_S)^2}{2^\lambda}$$

In ***Game 3*** we guess $\mathcal{B}$ and $\mathcal{C}$ as well as the matching sessions, that $\mathcal{A}$ attacks and abort if the guess is wrong. Thus, we find:

$$\Pr[\text{break}_2] \leq n_P^2 \cdot n_S^2 \cdot \Pr[\text{break}_3]$$

In ***Game 4*** we replace the randomness used for key-encapsulation with $\mathcal{C}$'s public key with true randomness if SEEC is used and don't do anything otherwise. To show that this replacement is sound, we initialize a SEEC-challenger for Σ and replace all of $\mathcal{B}$'s computations of GenRand in other sessions with invocations of GenRand′, use the getKey- and getRandomness-oracles to answer any corruption-queries by $\mathcal{A}$ and replace the GenRand-call for the encapsulation with an invocation of the Challenge-oracle. If the challenge-bit b is zero, then the encapsulation-randomness is computed via GenRand and we are in *Game 3*. Otherwise, it is a true random value, and

we are in *Game 4*. Since $\mathcal{A}$ is not allowed to corrupt both $\mathcal{B}$'s static key and $\mathcal{B}$'s ephemeral key in the target-session, this reduction follows the rules of the SEEC-experiment and we find:

$$\Pr[\text{break}_3] \leq \Pr[\text{break}_4] + \mathsf{Adv}^{SEEC}_{\Sigma,\mathcal{A}'}(1^\lambda)$$

Alternatively, if SEEC is not used we don't do anything here and thus trivially find:

$$\Pr[\text{break}_3] = \Pr[\text{break}_4]$$

In ***Game 5*** we replace the key $k_{\mathcal{C}}$ encapsulated in the ciphertext $c_{\mathcal{C}}$ for $\mathcal{C}$'s public key with a uniform random key. To show that this replacement is sound we initialize an ICC-CCA-challenger for XKEM and replace $\mathcal{C}$'s static public key with the challenge public key. Whenever $\mathcal{C}$ needs to perform a decapsulation we use the decapsulation-oracle instead which will by definition occur at most n_S times. If the ciphertext $c_{\mathcal{C}}$ is replayed in later sessions, we use the same value for $k_{\mathcal{C}}$ there, that we use for the current session. This substitution is valid since the encapsulation of the challenge-ciphertext uses true randomness by *Game 4* and the static secret key is, by the definition of this case, not corrupted and only used for decapsulations that can easily be replaced with oracle-calls, If the challenge-bit is zero, then all operations are still performed as before, and we are in *Game 4*. Otherwise, the key has been replaced with an independent random value and we are in *Game 5*. Thus, we find:

$$\Pr[\text{break}_4] \leq \Pr[\text{break}_5] + \mathsf{Adv}^{\mathsf{ICC\text{-}CCA}}_{XKEM,\mathcal{A}',n_S}(1^\lambda)$$

In ***Game 6*** we replace all outputs of the (implicit) hash-object after inputting $k_{\mathcal{C}}$, the previously replaced key, with random values. To show that this replacement is sound we initialize a PRHO-challenger for HO and replace $\mathcal{C}$'s direct use of HO with the oracles in the following way: Whenever $\mathcal{C}$ starts a session and would normally initialize a hash-object, she will instead call Create and use the returned identifier i_{HO} for all oracle invocations in that session. Whenever $\mathcal{C}$ would normally use the input/finalize functions of HO she will instead invoke the In/Fin oracle, with one exception: Whenever $\mathcal{C}$ would normally use the input/finalize functions of HO with the previously replaced value $k_{\mathcal{C}}$, she will instead invoke $\mathsf{Rand}(i_{\mathsf{HO}}, \text{false/true})$ including any later session that reuses that key (due to replays). This substitution is valid since $k_{\mathcal{C}}$ is an independent random value by *Game 5*. Therefore, If the challenge bit b is zero, this is a purely conceptual change and we are in *Game 5*. Otherwise, all outputs after inputting $k_{\mathcal{C}}$ get replaced with independent random values and we are in *Game 6*. Thus:

$$\Pr[\text{break}_5] \leq \Pr[\text{break}_6] + \mathsf{Adv}^{PRHO}_{\mathsf{HO},\mathcal{A}'}(1^\lambda)$$

In ***Game 7*** we terminate the execution of all sessions that recreate the state of the hash-object after $k_{\mathcal{C}}$ has been fed into it in the same way an honest party would terminate the execution after receiving an invalid AEAD-ciphertext. This might happen as the result of the adversary replaying messages including KEM-ciphertexts in a later session to then extract information about the target-session from that other session. Let $k^{\dagger}$ be the first AEAD-key that is generated from the hash-object after feeding $k_{\mathcal{C}}$ into it. To show that this replacement is sound we use two sub-games during which we will first guess the index of the first session in which the adversary sends a successfully forged ciphertext and then argue that this ciphertext can be used to break the authenticity of the AEAD-scheme. Let *Game 7.0* = *Game 6*.

In ***Game 7.1*** we guess the index of the first session in which the adversary creates a valid ciphertext for $k^{\dagger}$ if there is any. Given that a party may be involved in up to n_s sessions of which one is the target-session and can therefore be ignored, we find:

$$\Pr\left[\text{break}_{7.0}\right] \leq (n_S - 1) \cdot \Pr\left[\text{break}_{7.1}\right]$$

In ***Game 7.2*** we terminate the execution of all sessions that recreated $k^{\dagger}$. To show that this replacement is sound, we initialize an authenticity-challenger for our AEAD-scheme and replace all encryptions under $k^{\dagger}$ with calls to the encryption-oracle and return the first valid ciphertext to the challenger. This substitution is valid, since $k^{\dagger}$ is random by *Game 6* and we know which ciphertext is the first valid forgery by *Game 7.1*, we furthermore know by *Game 2*, *Game 1* and the definition of the protocol that all of $\mathcal{C}$'s sessions use a different ephemeral public key, that therefore h is different in every session and that since h is used as associated data for the AEAD-scheme that all honestly generated ciphertexts would have to decrypt successfully with different associated data (which would be accepted by the authenticity-game as a successful attack). If the ciphertext by the adversary is invalid, then we are terminating in either case and this change is perfectly undetectable. Otherwise, we win the authenticity-game for the AEAD-scheme and thus find:

$$\Pr\left[\text{break}_{7.1}\right] \leq \Pr\left[\text{break}_{7.2}\right] + \mathsf{Adv}^{\mathsf{auth}}_{AEAD,\mathcal{A}'}\left(1^{\lambda}\right)$$

Combining the sub-games we get:

$$\Pr\left[\text{break}_{7.0}\right] \leq \Pr\left[\text{break}_{7.2}\right] + (n_S - 1) \cdot \mathsf{Adv}^{\mathsf{auth}}_{AEAD,\mathcal{A}'}\left(1^{\lambda}\right)$$

We note that the correctness of the guess in *Game 7.1* only matters for *Game 7.2* and there only in the event of an authenticity-break of the AEAD-scheme, which means that it does not affect the following games and by

defining *Game 7* := *Game 7.2* we find:

$$\Pr[\text{break}_6] \leq \Pr[\text{break}_7] + (n_S - 1) \cdot \mathsf{Adv}^{\mathsf{auth}}_{AEAD,\mathcal{A}'}(1^\lambda)$$

In ***Game 8*** we guess the session key $\hat{k}$ for which $\mathcal{A}$ will guess the challenge-bit. There are by definition n_K session keys in total, which forms a lower bound of $\frac{1}{n_K}$ for guessing correctly and we find:

$$\Pr[\text{break}_7] \leq n_K \cdot \Pr[\text{break}_8]$$

In ***Game 9*** we use the remaining adversarial advantage to win the IND$-CPA-game for the AEAD-scheme with the same advantage. In order to do so we initialize an IND$-CPA-challenger and replace all encryptions using $\hat{k}$ with oracle invocations of Enc. This substitution is valid as $\hat{k}$ is random and independent by *Game 7* and because the AEAD challenge-bit and the stage-bit are sampled from the same distribution. We note that by the argument given in *Game 6*, $\mathcal{C}$ will not use a key that is not derived from her ephemeral key to encrypt any message, and that by *Game 7* all keys that she uses to encrypt messages in the target-session are random and independent of the keys in all other sessions. We then forward $\mathcal{A}$'s guess of the stage-bit to the AEAD-challenger and win if and only if $\mathcal{A}$ wins, giving us:

$$\Pr[\text{break}_8] = \Pr[\text{break}_9] = \mathsf{Adv}^{\mathsf{IND\$\text{-}CPA}}_{AEAD,\mathcal{A}'}(1^\lambda)$$

By summarizing these losses and applying Lemma 1 we find the adversarial advantages stated in Theorems 17-20. □

The last two of the six confidentiality theorems deal with uncorrupted ephemeral keys and are largely analogous to the previous four besides that.

Theorem 21. Π *achieves confidentiality in stage #E if both the initiator and the responder have at least one uncorrupted key and both are honest partners with:*

$$\mathsf{Adv}^{fACCE.conf_E}_{\Pi,\mathcal{A}}(1^\lambda) \leq \tfrac{4(n_P \cdot n_S)^2}{2^\lambda} + \mathsf{Adv}^{\mathsf{CollRes}}_{\mathsf{H},\mathcal{A}'}(1^\lambda) + n_P^2 \cdot n_S^2 \cdot (2 \cdot \mathsf{Adv}^{SEEC}_{\Sigma,\mathcal{A}'}(1^\lambda) + EKEM.\delta + \mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{EKEM,\mathcal{A}',1}(1^\lambda) + \mathsf{Adv}^{\mathsf{PRHO}}_{H,\mathcal{A}'}(1^\lambda) + n_K \cdot \mathsf{Adv}^{\mathsf{IND\$\text{-}CPA}}_{AEAD,\mathcal{A}'}(1^\lambda)),$$

where n_K is the total number of session-keys used in a session and n'_K refers to the total number of session keys that have been used before the key encapsulated in EKEM has been put into the hash-object.

Theorem 22. Π' *achieves confidentiality in stage* $\#E$ *if neither the initiators nor the responder's ephemeral keys are uncorrupted, and both are honest partners with:*

$\mathsf{Adv}^{fACCE.conf_E}_{\Pi,\mathcal{A}}(1^\lambda) \leq \frac{4(n_P \cdot n_S)^2}{2^\lambda} + \mathsf{Adv}^{\mathsf{CollRes}}_{\mathsf{H},\mathcal{A}'}(1^\lambda) + n_P^2 \cdot n_S^2 \cdot (EKEM.\delta + \mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{EKEM,\mathcal{A}',1}(1^\lambda) + \mathsf{Adv}^{\mathsf{PRHO}}_{H,\mathcal{A}'}(1^\lambda) + n_K \cdot \mathsf{Adv}^{\mathsf{IND\$\text{-}CPA}}_{AEAD,\mathcal{A}'}(1^\lambda))$,

where n_K *is the total number of session-keys used in a session and* n'_K *refers to the total number of session keys that have been used before the key encapsulated in EKEM has been put into the hash-object.*

The proofs for these theorems are largely analogous to the one before, except that they require two applications of SEEC (one on either peer).

Proof. We use game hopping to show the statements. The following proof assumes without loss of generality, that the ephemeral public key is sent by the initiator. We use this convention here as it is the case in all PQNoise base-patterns as well as almost all sensible PQNoise patterns in general and because simply relabeling initiator and responder is sufficient for the complimentary case.

Game 0 refers to the original fACCE-game.

In ***Game 1*** we abort if there is ever a collision of the ephemeral entropy. The probability of this is given by a birthday-bound and we find:

$$\Pr[\text{break}_0] \leq \Pr[\text{break}_1] + \frac{4(n_P \cdot n_S)^2}{2^\lambda}$$

In ***Game 2*** we guess $\mathcal{I}$ and $\mathcal{R}$ as well as the matching sessions, for which $\mathcal{A}$ guesses a stage-bit and abort if we guess wrong. Thus, we find:

$$\Pr[\text{break}_1] \leq n_P^2 \cdot n_S^2 \cdot \Pr[\text{break}_2]$$

In ***Game 3*** we replace the randomness used for generation of $\mathcal{I}$'s ephemeral keypair with true randomness if SEEC is used and don't do anything otherwise. To show that this replacement is sound, we initialize a SEEC-challenger for Σ and replace all of $\mathcal{I}$'s computations of $\mathsf{GenRand}$ in other sessions with invocations of $\mathsf{GenRand}'$, use the getKey- and $\mathsf{getRandomness}$-oracles to answer any corruption-queries by $\mathcal{A}$ and replace the $\mathsf{GenRand}$-call for the encapsulation with an invocation of the $\mathsf{Challenge}$-oracle. If the challenge-bit b is zero, then the encapsulation-randomness is computed via $\mathsf{GenRand}$ and we are in *Game 2*. Otherwise, it is a true random value, and we are in *Game 3*. Since $\mathcal{A}$ is not allowed to corrupt both $\mathcal{I}$'s static key and $\mathcal{I}$'s ephemeral key in the

target-session, this reduction follows the rules of the SEEC-experiment and we find:

$$\Pr[\text{break}_2] \leq \Pr[\text{break}_3] + \mathsf{Adv}^{SEEC}_{\Sigma,\mathcal{A}'}(1^\lambda)$$

Alternatively, if SEEC is not used we don't do anything here and thus trivially find:

$$\Pr[\text{break}_2] = \Pr[\text{break}_3]$$

In ***Game 4*** we replace the randomness used for key-encapsulation with $\mathcal{I}$'s public key with true randomness if SEEC is used and don't do anything otherwise. To show that this replacement is sound, we initialize a SEEC-challenger for Σ and replace all of $\mathcal{R}$'s computations of GenRand in other sessions with invocations of GenRand′, use the getKey- and getRandomness-oracles to answer any corruption-queries by $\mathcal{A}$ and replace the GenRand-call for the encapsulation with an invocation of the Challenge-oracle. If the challenge-bit b is zero, then the encapsulation-randomness is computed via GenRand and we are in *Game 3*. Otherwise, it is a true random value, and we are in *Game 4*. Since $\mathcal{A}$ is not allowed to corrupt both $\mathcal{R}$'s static key and $\mathcal{R}$'s ephemeral key in the target-session, this reduction follows the rules of the SEEC-experiment and we find:

$$\Pr[\text{break}_3] \leq \Pr[\text{break}_4] + \mathsf{Adv}^{SEEC}_{\Sigma,\mathcal{A}'}(1^\lambda)$$

Alternatively, if SEEC is not used we don't do anything here and thus trivially find:

$$\Pr[\text{break}_3] = \Pr[\text{break}_4]$$

In ***Game 5*** we replace the key k_E encapsulated in the ephemeral ciphertext c_E with a uniform random key. To show that this replacement is sound we initialize an IND-CCA-challenger for EKEM and replace the ephemeral public key with the challenge public key. This substitution is valid since in the case of a correctly transmitted c_E the ephemeral keypair is honestly generated with true randomness by *Game 3* and the encapsulation of the challenge-ciphertext uses true randomness by *Game 4*. In case the c_E that $\mathcal{I}$ receives ($=: c'_E$)) is different from the one $\mathcal{R}$ sent, then the two sessions no longer match, thus the freshness bit is set to zero for all following stages which the adversary can no longer attack in order to win.

The initiator will in this case use the decapsulation-oracle once to decapsulate c'_E and continue the protocol with the decapsulated key. If the challenge-bit is zero, then all operations are still performed as before, and we are in *Game 4*. Otherwise, the key has been replaced with an independent random

value and we are in *Game 5*. Thus, we find:

$$\Pr[\text{break}_4] \leq \Pr[\text{break}_5] + \mathsf{Adv}^{\mathsf{ICC\text{-}CCA}}_{EKEM,\mathcal{A}',1}(1^\lambda)$$

In ***Game 6*** we replace all outputs of the (implicit) Hash-object after inputting the previously replaced key with random values. To show that this replacement is sound we initialize a PRHO-challenger for H and replace $\mathcal{C}$'s direct use of H with the oracles in the following way: Whenever $\mathcal{C}$ starts a session and would normally initialize a hash-object, she will instead call Create and use the returned identifier i_{H} for all oracle invocations in that session. Whenever $\mathcal{C}$ would normally use the $\mathsf{input}/\mathsf{finalize}$ functions of H with the previously replaced value k_E, she will instead invoke the In/Fin oracle, except: This substitution is valid since k_E is an independent random value by *Game 5*. Therefore, if the challenge bit b is zero, this is a purely conceptual change and we are in *Game 5*. Otherwise, all outputs after inputting $k_{\mathcal{C}}$ get replaced with independent random values and we are in *Game 6*. Thus:

$$\Pr[\text{break}_5] \leq \Pr[\text{break}_6] + \mathsf{Adv}^{PRHO}_{\mathsf{H},\mathcal{A}'}(1^\lambda)$$

In ***Game 7*** we guess the session key $\hat{k}$ for which $\mathcal{A}$ will guess the challenge-bit if $c'_E = c_E$. There are by definition n_K session keys in total, which forms a lower bound of $\frac{1}{n_K}$ for guessing correctly and we find: If $c'_E \neq c_E$ there is only one stage $\mathcal{A}$ can attack, in which case this guess becomes trivial and since $1 \leq n_K$ we find:

$$\Pr[\text{break}_6] \leq n_K \cdot \Pr[\text{break}_7]$$

In ***Game 8*** we use the remaining adversarial advantage to win the $\mathsf{IND\$\text{-}CPA}$-game for the AEAD-scheme with the same advantage. In order to do so we initialize an $\mathsf{IND\$\text{-}CPA}$-challenger and replace all encryptions using $\hat{k}$ with oracle invocations of Enc. This substitution is valid as $\hat{k}$ is random and independent by *Game 7* and because the AEAD challenge-bit and the stage-bit are sampled from the same distribution. We then forward $\mathcal{A}$'s guess of the stage-bit to the AEAD-challenger and win if and only if $\mathcal{A}$ wins, giving us:

$$\Pr[\text{break}_7] = \Pr[\text{break}_8] = \mathsf{Adv}^{\mathsf{IND\$\text{-}CPA}}_{AEAD,\mathcal{A}'}(1^\lambda)$$

By summarizing these losses and applying Lemma 1 we find the adversarial advantage stated in Theorems 21 and 22. □

Using these results we can then easily set up the fACCE-table for any given PQNoise protocol: The authenticity-results can be taken as they are.

For confidentiality the lowest value that achieves security is the relevant one. For the fundamental PQNoise patterns this gives the results presented in Table 5.2.

Previous analysis [DRS20] of Noise used a version of the fACCE model that presented its end-results in a slightly different way, that essentially combined some rows of the version we are using into one. In particular its notions for authenticity and replay-protection did not distinguish whether the peer's ephemeral or static key was uncorrupted and its „eck“ notion was essentially defined as the first stage in which confidentiality was achieved in all settings where each party had at least one uncorrupted secret. Because of this we had to perform some interpretation of those results that resulted in Table 5.3. When we compare the two tables, we can see that PQNoise ends up with the same security as classical Noise eventually.

5.5. Implementation

Angel implemented the PQNoise as an extension of the "nyquist" implementation of Noise by Angel in the Go programming language [Ang, ADH+22b]. As underlying instantiation of all KEMs he used Kyber-768 [BDK+18, ABD+21]; specifically, the highly optimized Go implementation in Cloudflare's Circl library [KFH19]. As Circl implements other post-quantum KEMs, including all parameter sets of Kyber, it would be easy to change to a different instantiation of the KEMs.

When comparing performance between PQNoise handshakes and corresponding Noise handshakes, PQNoise is *computationally*, i.e., in terms of CPU cycles, more efficient. This is not surprising, because Kyber-768 is considerably faster than X25519-based DH key exchange in Noise. For example, on an Intel Xeon E-2124 (Coffee Lake) CPU, eBACS [BL] reports 125 303 cycles for X25519 key generation and 135 390 cycles for X25519 shared-key computation; on the same CPU eBACS reports only 39 881 cycles for Kyber-768 key generation, 53 841 cycles for encapsulation, and 42 281 cycles for decapsulation. On other recent 64-bit CPUs the absolute numbers differ, but the big picture is similar: Kyber-768 outperforms X25519 in terms of cycle counts. [ADH+22b]

However, this advantage in computational performance does not mean that handshake times for PQNoise are faster than in Noise. In fact, all cryptography used in Noise or in (our instantiation of) PQNoise is so fast that handshake times are largely determined by data transmission, and this is where two disadvantages of PQNoise kick in: first, post-quantum KEMs have much

Table 5.2.: Security of the fundamental PQNoise patterns. Values of the form x/∞ mean that the security is achieved in stage x if the party/parties that doesn't/don't use a static KEM still uses a static SEEC-key and never if that is not the case. We don't provide separate rows for authenticity without SEEC, as the $s_{\mathcal{I}}$, $e_{\mathcal{R}}/e_{\mathcal{I}}$, $s_{\mathcal{R}}$-cases are identical, and the $s_{\mathcal{I}}$, $s_{\mathcal{R}}$-cases are trivially insecure and have those rows dropped entirely if SEEC is not used.

Security	Uncorr.	N	NN	NK	NX	KN	KK	KX	XN	XK	XX	IN	IK	IX
Confidentiality	$e_{\mathcal{I}}$, $e_{\mathcal{R}}$	∞	2	2	2	2	2	2	2	2	2	2	2	2
	$e_{\mathcal{I}}$, $s_{\mathcal{R}}$	1	$2/\infty$	1	2	$2/\infty$	1	2	$2/\infty$	1	2	$2/\infty$	1	2
	$s_{\mathcal{I}}$, $e_{\mathcal{R}}$	∞	$2/\infty$	$2/\infty$	$2/\infty$	2	2	2	2	2	2	2	2	2
	$s_{\mathcal{I}}$, $s_{\mathcal{R}}$	$1/\infty$	$2/\infty$	$1/\infty$	$2/\infty$	$2/\infty$	1	2	$2/\infty$	1	2	$2/\infty$	1	2
Confidentiality	$e_{\mathcal{I}}$, $e_{\mathcal{R}}$	∞	2	2	2	2	2	2	2	2	2	2	2	2
(Without SEEC)	$e_{\mathcal{I}}$, $s_{\mathcal{R}}$	1	∞	1	3	∞	1	3	∞	1	3	∞	1	3
	$s_{\mathcal{I}}$, $e_{\mathcal{R}}$	∞	∞	∞	∞	2	2	2	4	4	4	2	2	2
Authenticity ($\mathcal{I}$)	$s_{\mathcal{I}}$, $e_{\mathcal{R}}$	∞	∞	∞	∞	3	3	3	5	5	5	3	3	3
	$s_{\mathcal{I}}$, $s_{\mathcal{R}}$	∞	∞	∞	∞	$3/\infty$	3	3	$5/\infty$	5	5	$3/\infty$	3	3
Authenticity ($\mathcal{R}$)	$e_{\mathcal{I}}$, $s_{\mathcal{R}}$	∞	∞	2	4	∞	2	4	∞	2	4	∞	2	4
	$s_{\mathcal{I}}$, $s_{\mathcal{R}}$	∞	∞	$2/\infty$	$4/\infty$	∞	2	4	∞	2	4	∞	2	4

Table 5.3.: Security of the fundamental Noise patterns, based on previous analysis [DRS20]. We remark that this is an interpretation of those results (as the presentation of our fACCE-results differs) and that parts of this table are only conjectured.

Property	Uncorrupted	N	NN	NK	NX	KN	KK	KX	XN	XK	XX	IN	IK	IX
Confidentiality	$e_{\mathcal{I}}$, $e_{\mathcal{R}}$	∞	2	2	2	2	2	2	2	2	2	2	2	2
	$e_{\mathcal{I}}$, $s_{\mathcal{R}}$	1	∞	1	2	∞	1	2	∞	1	2	∞	1	2
	$s_{\mathcal{I}}$, $e_{\mathcal{R}}$	∞	∞	∞	∞	2	2	2	3	3	3	2	2	2
	$s_{\mathcal{I}}$, $s_{\mathcal{R}}$	∞	∞	∞	∞	∞	1	∞	∞	∞	∞	∞	1	∞
Authenticity ($\mathcal{I}$)	$s_{\mathcal{I}}$, $e_{\mathcal{R}}$	∞	∞	∞	∞	3	3	3	3	3	3	3	3	3
	$s_{\mathcal{I}}$, $s_{\mathcal{R}}$	∞	∞	∞	∞	∞	1	∞	∞	∞	∞	∞	1	∞
Authenticity ($\mathcal{R}$)	$e_{\mathcal{I}}$, $s_{\mathcal{R}}$	∞	∞	2	2	∞	2	2	∞	2	2	∞	2	2
	$s_{\mathcal{I}}$, $s_{\mathcal{R}}$	∞	∞	∞	∞	∞	2	∞	∞	∞	∞	∞	2	∞

Table 5.4.: Median handshake times in ms of KK and XX Noise patterns and their pqKK and pqXX counterparts in PQNoise.

	Fast network		Slow network	
	Init.	**Resp.**	**Init.**	**Resp.**
KK	16.35	0.42	98.73	0.41
pqKK	16.07	0.25	100.28	0.27
XX	16.02	16.1	98.47	98.6
pqXX	31.83	16.1	199.31	100.36

larger public keys and ciphertexts than (pre-quantum) ECDH. Second, in some scenarios KEM-based AKE requires more round trips to achieve the same security. In order to investigate how these two factors influence real-world handshake performance, [ADH+22b] considers the KK and the XX handshake patterns from Noise together with their pqKK and pqXX counterparts from PQNoise. While both patterns eventually achieve mutual authentication, for the KK and pqKK patterns the number of round trips is the same, while for the pqXX pattern an additional message from the responder is required compared to XX. The benchmarks were run on a machine with two Intel Xeon Gold 6230 CPUs and 196 GB of RAM. The experimental setup uses the Linux kernel's network-emulation features and is largely following the setup used in [PST20] and [SSW20]. For each of the patterns 1000 measurements of the time it takes to perform a handshake were taken, independently for initiator and responder. Theses measurements were performed once over a fast network (1000 Mbit throughput, 31.1ms round-trip latency) and once over a slow network (10 Mbit throughput, 195.6ms round-trip latency). The results are listed in Table 5.4. The KK and pqKK responder times do not include any network communication – after receiving the first handshake message from the initiator, the responder can perform all computations without having to wait for a further message. These times thus show the computational advantage of Kyber-768 over X25519. We also see that increased message sizes in PQNoise do not have a major influence on performance in this TCP/IP-based scenario. What *does* matter is the additional protocol message in pqXX compared to XX: as expected, the initiator times are slower by pretty exactly the half-roundtrip network latency. [ADH+22b]

5.6. Conclusion

By designing and analyzing PQNoise in a generic way that is not tied to a specific pattern, we were not only able to provide post-quantum protocols for all standard Noise-settings but can even provide hard security-statements for Noise-protocols that use other primitives besides KEMs, such as hybrid versions. Our approach is furthermore sufficiently generic to be applicable to generically proving the use of these other primitives as well, allowing for a much more comprehensive analysis of all possible Noise-patterns than what exists at the moment.

On top of that our implementation demonstrates the practicability of post-quantum key-exchanges in a wide variety of settings. We remark here that providing a post-quantum version of Noise essentially provides a solution for all applications that need key-exchanges that are requiring neither backwards-compatibility nor crypto-agility; naturally the former is not a problem that can be solved generically and the latter is a property whose desirability is getting called increasingly into question.

6. PQC in Space

This chapter is largely based on the study "An Asymmetric-based Post-Quantum Cryptographic Protocol for Space Missions", co-written with Andreas Hülsing and Tanja Lange in 2024 for the European Space Agency (ESA). I gave an invited talk about this work at the Security for Space Systems conference 2024.

6.1. Introduction

Communication between satellites and mission control centers is currently protected via the Space Data Link Security Protocol (SDLS). At its core this protocol uses symmetric algorithms (most importantly symmetric encryption) to secure communication.

Key-updates are possible in SDLS, but they too are only using symmetric primitives. This has the advantage that they are not immediately vulnerable to quantum-computers, but the major downsides that a key-update cannot recover security after a key-compromise if the adversary receives a network-transcript of the key-update, that the approach does not scale well to decentralized networks, and that it limits operational efficiency, especially in federated operations.

The goal of this work was therefore to design a key-update/establishment protocol that relies on asymmetric algorithms for authenticity and confidentiality. A slightly unusual aspect of this goal is that the necessary protocol for this is not just a handshake-phase of a more complex protocol that is only ever run in conjunction with a data-transmission protocol, but actually independent.

Because of the looming threat that quantum-computers may in the not so far future be able to break all asymmetric primitives that are currently in widespread use, the protocol should include other, so-called "post-quantum" algorithms that are designed to withstand that threat. The primary source for post-quantum algorithms is the post-quantum competition run by NIST, that recently selected four winners, that are bound to be standardized. Beyond these winners there also a handful of more specialized algorithms that were

standardized by the IETF as well as algorithms that have not (yet) been chosen by NIST but were declared to be acceptable by various European government bodies or are in widespread use. (See Sections 6.5.1 and 6.5.2 for more details.)

6.2. Background

We refer to Section 2.2 for discussions of general cryptographic concepts, such as hash functions, Message Authentication Codes (MACs), Authenticated Key Exchanges (AKE), Diffie-Hellman Key Exchanges (DHKX), and Key Encapsulation Mechanisms (KEMs). For a more complete discussion of Noise, we refer to Section 5.1 and will only discuss the minor extensions to Noise that we use in this chapter here. Namely we add a small number of new tokens and annotations for tokens:

- `sig` refers to a signature under the sender's long-term key whose message is a hash of the full transcript of the handshake up to that point.
- `s'` is an extension that we propose in this work. It refers to an updated long-term public-key that is supposed to replace the old long-term key of the sending party after the successful completion of the handshake.
- `X[optN]`, where `X` is one of the tokens above and `N` is an integer, is a second extension that we propose in this work. We use it to indicate that the protocol can run as if the token `X` was part of the pattern or as if it was not there, the decision occurring at runtime. The integer `N` is simply used to separate multiple independent optional components.
- `confirm` is an extension that indicates a key confirmation message which sends an AEAD ciphertext for an empty message to prove knowledge of the current session key. (In Noise this is given by an arrow without tokens, but we make this more explicit.)

6.3. Requirements & Constraints

In this section we discuss the requirements and constraints that the use-case places upon the protocol, beginning with the former.

6.3.1. Requirements

We identified the following requirements in close consultation with experts from ESA:

Authenticity

Authenticity refers to the property that parties can be certain that they are interacting with the party that they think they are interacting with.

In some missions, where security is of high priority, authenticity has to be bidirectional to ensure that both the satellite and mission control are certain of each other's identity.

In some other missions authenticating the satellite to mission control might not be necessary, because the control center knows where the satellite is supposed to be and would use communication channels that are physically very narrow. In any case this is a per-mission property and thus does not have to be negotiated at runtime.

Confidentiality

Confidentiality refers to the property that the content of an interaction remains secret except to the authorized parties. In the context of a key-exchange this content refers to the resulting key being only known to the parties involved in the exchange, not anyone else, including active or passive network-attackers.

Confidentiality of the payload-data is usually a security goal but sometimes less important than authenticity and availability.

For the purposes of a key-update where a key derived from the shared secret is then used in SDLS this however means that since the knowledge of the shared secret is also used to authenticate a party, full confidentiality of that secret is required regardless.

We furthermore note that confidentiality is desirable under all circumstances, which includes before and after any potential compromises.

Availability

In the given setting, availability refers to the ability to communicate with the satellite.

As is common in many settings availability is generally very important here, a very short-term loss might however be tolerable. With regards to

DoS-protection ESA does not see a need to consider simultaneous connection-attempts but notes that continuous attempts to authenticate/connect are in scope.

Hybrid Security

Hybrid Security is the property that a protocol uses multiple algorithms to instantiate a primitive in a way such that the protocol remains secure even if a subset of the algorithms turns out to be insecure. While relatively few traditional protocols have done this, the comparative novelty of most post-quantum schemes makes their combination with a more established, compact, and highly performant, but pre-quantum scheme a popular choice for protocols. In line with a request by ESA, we took that approach for all post-quantum primitives in the key-update protocol.

The easiest way to achieve this is through the use of hybrid primitives, for example KEMs that are a combination of a lattice-based KEM and a KEM based on an elliptic-curve Diffie-Hellman key-exchange. This approach has several advantages over using the same primitive with different instantiations multiple times: It simplifies the protocol and its analysis, because the added complexity is contained in a sub-component that is not further decomposed in any case. It allows for greater flexibility when choosing or replacing concrete primitives, because the analysis depends less on the concrete instantiations. The main disadvantage of this approach is that it requires the definition of the hybrid primitive, though for the primitives that we consider relevant here (KEMs and Signatures), this is not a real problem, as there are established combiners that we will discuss in Section 6.5.4.

Adversaries

Considering the importance and cost of satellites the attacker-model for all properties has to include the full range of possibilities, from nation-states to individuals.

The most popular security-levels for post-quantum schemes are NIST's level I, III, and V, which correspond to the security of AES with 128, 192, and 256 bit keys. All of the cryptographic algorithms that we considered provide parameters for these security levels and are generally suitable for the full range of attackers, though with different security-margins; we thus design the protocol as agnostic of the security-level and analyze it with instantiations on all three levels.

For denial-of-service attacks in particular ESA did not require protection against connection attempts of multiple actors but did note that multiple connection attempts of a single actor should be considered.

We note that insofar availability is concerned the best that can be achieved on the protocol level is to avoid the introduction of vulnerabilities that allow an adversary to exhaust system resources with little adversarial effort, e.g., by filling up memory with initial connection attempts.

Ciphersuite negotiation vs. fixed choices

We say that a protocol is cryptographically opinionated if it prescribes certain algorithms without any option to negotiate others at runtime. The opposite of this is where some portfolio of permitted algorithms is defined and the ciphersuite is negotiated as part of the protocol. The latter has the downside that attackers might try to interfere with the negotiation process in a so-called downgrade attack; the former has the downside that weak defaults affect all users and that the full protocol may have to be updated or replaced if a component of it has to be replaced.

ESA indicated that they did not require ciphersuite-negotiation at runtime, and were willing to accept the need for a protocol-update in case a component needs to be replaced. We thus chose to design a cryptographically opinionated protocol for the greater simplicity and the reduced risk of downgrade-attacks.

Updateable Long-term keys

Besides the fairly standard previously listed requirements we also identified one unusual requirement that was found to be potentially desirable in follow-up discussions, namely the ability to upgrade the long-term keys of the parties:

The Requirement to replace communication keys indicates a concern that short-term keys could become corrupted and should thus be replaced. In light of that concern we should however also ask the question for why this is not a concern for long-term keys, considering that they are stored on the same devices. In light of this we decided to enable their updates as well.

In general there are three ways to perform such an update of a long-term key:

- As part of a separate protocol,
- as a part of the key-update mechanism that is always run, and

- as a part of the key-update mechanism that is not always run.

The first option has the disadvantage that it would require the implementation and analysis of another protocol, which might increase the attack-surface.

The second option is the most obvious one, but it essentially turns the long-term keys into short-term keys as well. This has several advantages, such as a significantly reduced attack-surface for replay-attacks, but may come at the cost of larger packets, due to the need to always include the next public-keys.

The third option is essentially a trade-off between typical packet-size and the advantages of single-use identity-keys.

Scalability

Currently SDLSP uses keys that are pre-established between the communicating parties. This works fine for communication between a satellite and mission-control, but is expected to run into severe scalability-issues once future plans to use SDLSP for communication between satellites in large constellations comes to pass.

To deal with this the protocol has to be based on asymmetric cryptography, and may only use a linear number of long-term-keys in the number of parties.

6.3.2. Constraints

The following constraints clarify the setting in which the protocol will be used. Especially, they point out limitations and constraints imposed on the solution.

Mission Classes

The primarily targeted missions use small to mid-sized commercial or institutional satellites in possibly larger constellations (more than 100 satellites), usually in low earth orbit.

The secondary targets are larger institutional or commercial satellites that use unclassified, standardized security functions.

CubeSat (cubes with 10cm side-length and no more than 2 kg weight) are out of scope for now but may be interesting later.

Governmental satellites that are used for classified data are out of scope.

With regards to the number of endpoints, the number of mission control systems on the ground is generally very limited (1, or 1 + redundancy), whereas the number of satellites may be comparatively high (in the hundreds or thousands).

Our assessment is that these constraints don't form meaningful issues as even the high end of the number of satellites is rather small compared to what many cryptographic protocols that run over the internet have to deal with.

Execution Environment

The IT infrastructure on the ground is generic and for the purposes of this project not meaningfully constrained.

The On-Board Computers (OBCs) that are still in use include some older chips such as GR740 and NG-Ultra; on new system the use of 8 MIPS (Millions of Instructions Per Second, not to be confused with the MIPS architecture) is likely acceptable. On old systems there is essentially no idle time, whereas newer systems may have a little bit of it (2%, which equates to the aforementioned 8 MIPS). Beyond that, there are no real additional constraints on energy-use.

The storage-capacity on the satellite may lie between 256 GB and 1 TB.

Our assessment is that in light of the performance of most post-quantum primitives, none of these environments appear to cause meaningful constraints on the design of the protocol.

Connection

The latency of a connection starts at 40-50 ms for low earth orbit and grows with the distance from there. For communication with Mars for example it lies in the 5-20 min range.

Packet sizes are around 1 KB. Packet-Fragmentation should preferably be avoided, but this is no hard requirement.

The Throughput depends on the concrete link:

- Sband: 1 Kbit/s - 100 Kbit/s
- Xband: up to 100s of Mbit/s, usually D/L only
- KA Band: up to several Gbit/s, mainly D/L
- Typical: 256kbps up, 2Mbps down. some uplinks 64kbps or 128kbps

The total communication-time over these links should be limited to 1 minute, because the contact time in low earth orbit lies in the 7-10-minute range.

In general these channel properties will not cause any issue for the protocols we design. Sizes depend on the cryptographic schemes used to instantiate them though. For most schemes these bandwidths are unlikely to cause significant issues, as the resulting packet sizes fall into the 3-5 KB range. However, there are exceptions when using the most conservative cryptographic schemes. In these cases, care has to be taken. Fragmentation is unavoidable for sending ephemeral keys for all systems, it can be avoided for KEM ciphertexts for some systems.

Public Key Infrastructure

A Public Key Infrastructure (PKI) can be used to easily update long-term keys of various parties.

As of now there are some standardization efforts, but nothing is implemented yet.

Because of this, we will assume in the following that long-term keys of the possible peers are pre-installed.

6.4. Design Considerations

In a deviation from the more traditional design, where keys are always established as part of the handshake of a data transmission protocol, the key-update/establishment protocol for SDLS will be a stand-alone protocol that can be run independently from the data-transmission protocol. Not only does this reduce the difficulty of an analysis, but it also decreases the complexity (and thereby the attack-surface) and allows to update keys less often which reduces the communication and computational load. Additionally, it minimizes the changes required on the SDLS protocol itself.

We deviate from Noise in the way we derive the final key because we are designing a key exchange rather than a full secure channel protocol which also transmits data. Firstly, Noise uses its final keys already during the handshake-phase to encrypt messages before receiving a key-confirmation from the peer. In order to ensure that the resulting keys of the key-update mechanism are truly fresh and can be used without any worries, we instead derive the final keys only once everything in the handshake has been completed by the party in question.

Furthermore, we combine the resulting pre-keys with the handshake-hash to improve the robustness against replay-attacks. Noise does not do this as it also covers the data communication layer. There, it includes a transcript-

hash as associated data in every ciphertext sent, so replays could not send any new data. In our case we treat SDLS as a standalone protocol to which we need to deliver a fresh key.

Lastly, we always insist on key-confirmations. This is an aspect of signature-based protocols that could potentially save half a round-trip at the expense of a much more fragile protocol that has to use "external" replay-protection and may in the most extreme cases even result in an unrecoverable loss of a shared communication-key with the satellite. We provide a much more detailed discussion of this property in Section 6.6.4 after we have established more technical context.

Considering that ESA stated that it is often infeasible to impersonate a satellite due to the physical properties of the datalink between orbit and ground, we also provide a variant that does not authenticate the responding party (here: the satellite) in exchange for a slightly more lightweight protocol. While the use of such a protocol requires careful consideration, a scenario in which a channel can be assumed to be partially authenticated can be an instance where the bandwidth-savings may be enough to justify the use of the weaker protocol. We thus provide such a version as an additional variant.

We remark here that such a version assigns trust to whoever provides the out-of-band authentication, which in the motivating example would be the ground-stations, not mission-control. Whether this trust is reasonable is not something that we can comment on, beyond *strongly* advising to thoroughly evaluate that question before any use of protocols that are only partially authenticated.

6.5. Available PQ KEMs and Signatures

In this section we provide an overview of the available post-quantum KEMs and signature schemes. We focus on NIST finalists as well as alternates and discuss their advantages and disadvantages. For a more detailed overview covering more background see [BDH+21], for NIST's motivation see [AAC+22].

6.5.1. KEMs

The following KEMs managed to become finalists or alternate candidates in the third round of the NIST-competition; the first three are what we consider to be the most relevant ones for this work.

Kyber: By now the NIST process finished and selected Kyber [SAB+22] as the new standard for key encapsulation. This makes Kyber an obvious

choice to consider. Kyber is a lattice-based scheme that (on a high level) follows the LPR blueprint [LPR10] to design a lattice-based PKE which is then turned into a KEM using the FO transform [HHK17]. Kyber has very good overall performance and reasonably small key- and ciphertext-sizes. The security of Kyber is based on the hardness of standard lattice problems over module-lattices. The wider community considers the security of Kyber well understood.

Classic McEliece: The McEliece cryptosystem dates back to the beginnings of public key cryptography. It is widely considered one of the most conservative KEM constructions. For that reason, the NIST candidate Classic McEliece [ABC+22] has been officially recommended by the German BSI, the French ANSSI and the Dutch National Cyber Security Center (NL-NCSC). The scheme has a very interesting performance characteristic: While public keys are massive, ciphertext size is far smaller than for any other PQ-KEM. Moreover, implementations are fast. The small ciphertexts can make Classic McEliece an interesting choice in settings where public keys can be transmitted out-of-band or ahead of time.

Frodo: FrodoKEM [NAB+20] is a lattice-based KEM that, like Kyber, follows the LPR + FO blueprint. The big difference to Kyber is that Frodo uses unstructured lattices instead of module lattices which is generally considered the more conservative choice. However, this comes at a cost. Frodo has significantly worse sizes compared to Kyber. Still, due to the conservative security, Frodo is approved by BSI/ANSSI/NL-NCSC.

Saber: The Saber [DKR+20] construction is extremely close to that of Kyber with the one major difference that it uses rounding instead of adding an error. This bases security on the Learning-With-Rounding (LWR) problem instead of the Learning-With-Errors (LWE) problem. Consequently, it is in all performance aspects quite similar to Kyber, while possibly somewhat more efficient on constrained devices. NIST's report stated as reason to select Kyber over Saber was that for LWR fewer cryptanalysis results were available than for LWE.

NTRU: The NTRU [CDH+20] cryptosystem is the oldest lattice-based encryption scheme. It is based on the NTRU problem, a problem over certain structured lattices. While performance is overall good, it is slightly worse than that of Kyber, especially key generation is somewhat slower which affects usage for ephemeral keys. The scheme has

seen a long history of cryptanalysis without any major attacks (this is in contrast to NTRUSign which was broken). Based on this, also the security of NTRU is widely considered well understood by now. Google announced [ISE22] that they are using NTRU to secure internal communications.

NTRU Prime: The general idea of NTRU Prime [BBC+20] is to move lattice-based schemes to a different kind of lattice for which the team claims that it provides less exploitable structure than other lattices. The NTRU Prime family contains two KEMs: Streamlined NTRU Prime (sntrup) and NTRU LPRime (ntrulpr). The former is a variant of NTRU over the NTRU Prime lattice, the latter is a variant of the LPR approach, similar to Kyber over the NTRU Prime lattice. Performance-wise the two schemes are similar to the originals with the originals being slightly larger (as NTRU Prime uses rounding), though ciphertexts are marginally larger because the system uses an extra 32 bytes for a key confirmation hash. NIST stated that they did not select NTRU Prime because there was insufficient evidence for the security advantages of the scheme. Streamlined NTRU Prime is used as default in OpenSSH [Op22].

SIKE: The SIKE scheme was a KEM-candidate in Round 4 of the NIST competition based on the isogeny problem with additional information between supersingular curves. However, SIKE has been broken [CD23, MMP+23, Rob23] in 2022 and the underlying problem has been shown to be solvable in polynomial time.

BIKE and HQC: NIST moved two more code-based KEMs into Round 4, namely BIKE [ABB+22] and HQC [AAB+22]. These use structured codes similar to the cyclotomic rings in structured lattices. The use of structured codes allows them to achieve significantly smaller public keys than McEliece at the expense of larger ciphertexts. The ratio of ciphertext and public key size matches that of systems based on lattices, both structured and unstructured, and the sizes are between those of structured lattices and those of FrodoKEM. Also, the security of these schemes is less understood compared to lattice-based schemes: Code-based schemes using structured codes have a somewhat troubled history, e.g., systems using cyclic codes (rather than quasi-cyclic codes) were broken. The codes used in these schemes have not been subject to an attack but are also more recent and fewer cryptanalysis results are known than for the lattice-based schemes.

In our theoretical analysis we focus on Kyber, McEliece, and Frodo. The reason is that the performance of Saber, NTRU, and NTRU Prime is close enough to that of Kyber to not make a huge difference in a theoretical analysis (note that when evaluating practical performance, the slower key generation of NTRU and sntrup may make a difference) [AAC+22]. Similarly, BIKE and HQC are similar in performance to structured lattice-based schemes (although noticeably larger) [AAC+22]. Therefore results would be similar to those for Kyber and will not open up different trade-offs. Frodo demonstrates what it costs to go to unstructured lattices. SIKE is broken [CD23] and there is no similarly small scheme in the NIST process.

An ever-reoccurring topic when it comes to lattice-based KEMs is the topic of patents. We are not patent lawyers and cannot comment on this issue. NIST announced that it has come to agreements with holders of patents that potentially concern Kyber. According to NIST, the agreement allows the royalty free use of Kyber as standardized by NIST. We leave it to patent lawyers to discuss any further issues related to patents.

We depict the sizes of private keys (sk), public keys (pk) and ciphertexts (ct) for the three systems in the reference implementation in Table 6.1 and Figure 6.1, for comparison the former also includes the pre-quantum elliptic-curve scheme X25519. Note that private keys often can be compressed down to a seed at the expense of somewhat slower decapsulation times, e.g., the key format of Classic McEliece supports compressed private keys of size 32 bytes.

Table 6.1.: Key- and Ciphertext-sizes, Estimated NIST security-level ("Sec") and probability of decapsulation-failure (δ) of various KEMs. X25519 is an elliptic-curve-based pre-quantum key-exchange whose characteristics are representative of those of elliptic-curve based KEMs, which are our recommendation for the fallback-scheme in hybrid KEMs. Were it not for quantum-attacks X25519 would likely be classified as a level-1 scheme.

Scheme	SK	PK	CT	Sec.	δ
X25519	32	32	32	-	0
Kyber-512	1632	800	768	1	2^{-139}
Kyber-768	2400	1184	1088	3	2^{-164}
Kyber-1024	3168	1568	1568	5	2^{-174}
mceliece348864	6492	261120	96	1	0
mceliece460896	13608	524160	156	3	0

Scheme	SK	PK	CT	Sec.	δ
mceliece6688128	13932	1044992	208	5	0
mceliece6960119	13948	1047319	194	5	0
mceliece8192128	14120	1357824	208	5	0
FrodoKEM-640	19888	9616	9720	1	$2^{-138.7}$
FrodoKEM-976	31296	15632	15744	3	$2^{-199.6}$
FrodoKEM-1344	43088	21520	21632	5	$2^{-252.5}$

6.5.2. Signatures

NIST PQC chose three signature-schemes and additionally standardized a stateful signature-scheme before that:

Dilithium: Dilithium [LDK+22] is NIST's primary choice for signatures. It is based on structured lattices, using module lattices as Kyber. Public keys and signatures are about the same size and are both somewhat larger than their KEM counterparts. It provides decent performance and size characteristics, without major downsides.

Falcon: NIST also recommends FALCON [PFH+22], another lattice-based signature scheme using structured lattices, for settings in which small signatures are important. While having the same overall characteristics as Dilithium, signatures in Falcon are significantly smaller, so much smaller that they are smaller than Kyber ciphertexts. The downside of the scheme is that its definition uses floating-point arithmetic which is not available on smaller platforms, and which is notoriously difficult to implement securely.

SPHINCS+: SPHINCS+ [HBD+22] is the third recommendation of NIST. The system is based on hash functions. While it is extremely conservative in its assumptions, even more so than established pre-quantum schemes, and its public keys are small, its signatures are most likely too large to be of any use in environments with limited bandwidth, such as communication with satellites. Despite this limitation, SPHINCS+ is built from components that may be useful for this project.

Beyond these signature-schemes that NIST choose during the post-quantum competition, and which follow the regular definition of signature schemes, there are also stateful hash-based signature-schemes that the IETF and NIST

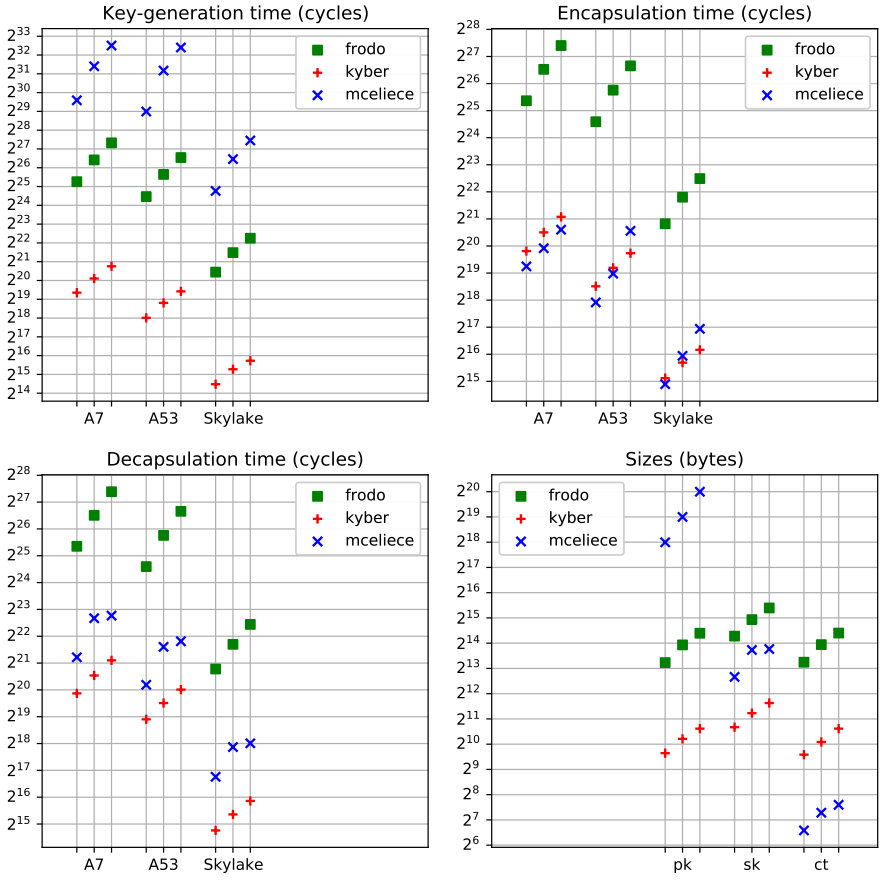

Figure 6.1.: Size- and Performance-characteristics of Frodo, Kyber and McEliece at NIST Level 1, Level 3 and Level 5 on ARM Cortex-A7, ARM Cortex-A53 and Intel Skylake CPUs using benchmarking data from eBACS.

have standardized and that are in the process of being standardized by ISO. They provide possibly interesting trade-offs and are already under consideration with ESA for the purpose of signing software:

XMSS: RFC 8391 [HBG+18] standardizes XMSS: Like SPHINCS+ it is a hash-based signature-scheme with extremely conservative assumptions (SPHINCS+ can in fact be seen as an extended version of XMSS), but unlike SPHINCS+ it has to maintain a state when signing: Each signing-operation updates that state and reusing an old state destroys the security of the entire scheme. Furthermore, the system fixes the number of signatures that can be made with one key, i.e., the state-update can only be performed a limited number of times (a typical limit would be 2^{20}). Signatures in XMSS are much smaller than in SPHINCS+, the exact length depends on the number of signatures the key can sign, and the performance is good.

LMS: RFC 8554 [MCF19] standardizes Leighton-Micali Hash-Based Signatures (LMS), another stateful hash-based signature scheme. The scheme is very similar to XMSS, with slightly larger signatures, a reliance on random oracles in its security-proof and about four times better overall performance.

WOTS: At their core SPHINCS+, XMSS and LMS are all schemes that are designed around the idea of using Winternitz One-Time-Signatures (WOTS) in a way that allows to sign more than one message. WOTS are hash-based signatures that can only be used to sign one message and become insecure if they are used twice. They have very small public keys and depending on the exact parameterization also quite small signatures and fast verification. As such we consider them a viable signature scheme too, as long as their main downside of only being able to sign once does not cause issues.

Beyond these options, the following algorithms also made it into the third round of the NIST-competition:

Rainbow: Rainbow [DCP+20] is a multivariate scheme that became a NIST-finalist. It shares the typical performance characteristic of multivariate schemes in that it offers very short signatures and very large public keys. However, Beullens [Beu22] showed that the design does not offer any advantages over the plain unbalanced oil-and-vinegar construction and broke the level-1 parameters.

GeMSS: GeMSS [CFM+20] is a multivariate scheme based on HFEv-. It has large public keys (between 0.3 and 3 megabytes) and small signatures. It was an alternate candidate in the NIST-competition and its security was degraded by [TPD21].

Picnic: Picnic [ZCD+20] is a scheme that like SPHINCS+ only relies on symmetric primitives. The schemes have similar properties and in the end, Picnic simply failed to demonstrate a clear advantage over its slightly more conservative competitor, causing it to lose the NIST-competition in which it was an alternate candidate.

NIST has started a new competition for post-quantum signatures and submissions were due in June 2023. We do not consider these submissions in scope as they have not received sufficient security analysis beyond some that are already completely broken.

In summary Dilithium and Falcon are the two primary options for a signature-scheme if bandwidth is a concern. The choice between them is largely a trade-off between higher bandwidth-use and higher implementation-difficulty. In case state-based schemes are acceptable, XMSS and LMS are possible options too that pay for their larger signatures with massively smaller public keys and radically fewer assumptions. The most extreme option is then to use WOTS directly which only allows to sign a single message.

Beyond these options, SPHINCS+ and Picnic have to be considered non-viable due to their large signatures, and Rainbow and GeMSS do not reach the claimed security levels. For these reasons we do not consider these options further for the key-update mechanism.

We depict the sizes of the NIST-winners and the elsewhere standardized stateful signature schemes in Table 6.2.

Table 6.2.: Estimated security-level, Key- and signature-sizes of various signature schemes. The security-level ("Sec") is given as NIST-level, where levels 1/3/5 are roughly equivalent to the security of AES 128/192/256 under quantum and classical attacks. Levels 2 and 4 are matched with collision-resistance of a hash function with 256 and 512 bits, respectively. The sizes of the WOTS+-keys are given assuming the use of key-compression. ECDSA is a pre-quantum scheme for cases where hybrid signatures are desirable; were it not for quantum-attacks it would likely be classified as a level-1 scheme.
(*) The stateful signature-schemes XMSS and LMS do not explicitly standardize the format of their secret-keys. The given sizes are assuming the use of pseudorandom key generation. Any efficient implementation will require an additional state with a size depending on the implementation and parameters used, typically in the order of kilobytes. Depending on the implementation, the whole state or most of it is non-secret information.

Scheme	SK	PK	Sig	Sec
Dilithium2	2544	1312	2420	1
Dilithium3	4016	1952	3293	3
Dilithium5	4880	2592	4595	5
Falcon-512	1281	897	666	1
Falcon-1024	2305	1793	1280	5
SPHINCS+-128s	64	32	7856	1
SPHINCS+-128f	64	32	17088	1
SPHINCS+-192s	96	48	16224	3
SPHINCS+-192f	96	48	35664	3
SPHINCS+-256s	128	64	29792	5
SPHINCS+-256f	128	64	49856	5
XMSS-SHA2_10_256	*64	64	2500	5
XMSS-SHA2_16_256	*64	64	2692	5
XMSS-SHA2_20_256	*64	64	2820	5
LMS_SHA256_M32_H15 with LMOTS_SHA256_N32_W4	*52	56	2664	5
LMOTS_SHA256_N32_W1	*52	56	8516	5
LMOTS_SHA256_N32_W2	*52	56	4292	5
LMOTS_SHA256_N32_W4	*52	56	2180	5
LMOTS_SHA256_N32_W8	*52	56	1124	5

Scheme	SK	PK	Sig	Sec
WOTS+(32,16)	32	32	2144	5
WOTS+(32,4)	32	32	4256	5
ECDSA-P256	32	32	64	-

6.5.3. Combining PQC with ECC

Requirement 6.3.1 asks to deploy the post-quantum algorithms in combination with "pre-quantum" public-key cryptography as also required by several European security agencies, including BSI and ANSSI. For the pre-quantum algorithms, elliptic curve cryptography (ECC) is preferable over RSA-based solutions due to their size. The common way to combine PQC and ECC algorithms is by the means of so-called combiners. In the following we discuss combiners for KEM and signature. Another detailed discussion of the topic can be found in ENISA's integration study on post-quantum cryptography [BHLR22].

6.5.4. KEM-Combiners

The goal of a KEM-combiner as considered in this work is to construct a KEM $\mathcal{S}$ from two KEMs $\mathcal{P}_1, \mathcal{P}_2$ such that $\mathcal{S}$ is secure as long as at least one out of $\mathcal{P}_1, \mathcal{P}_2$ is secure. For security we require security against active attacks, which is called ciphertext-INDistinguishability under Chosen Ciphertext Attacks (IND-CCA). Consequently, we describe our protocol with respect to a single KEM that can then be instantiated with any secure KEM, including a KEM that is obtained via any secure KEM combiner.

It turns out that the design of a robust KEM combiner is not as straightforward as one may think. There are several pitfalls possible as observed by Giacon, Heuer, and Poettering [GHP18]. However, there also exist several efficient and secure solutions. The combiners commonly run the two schemes $\mathcal{P}_1, \mathcal{P}_2$ independently and only combine their data objects afterwards for derivation of the shared secret. The most simple and still secure construction takes the shared secrets k_1, k_2 and the ciphertexts c_1, c_2 of $\mathcal{P}_1$, and $\mathcal{P}_2$, respectively, and computes the final shared key as

$$k = \mathcal{H}((k_1 \| k_2), (c_1 \| c_2)),$$

where $\mathcal{H}$ is a key-derivation function.

This construction can be proven secure modeling $\mathcal{H}$ as a random oracle, a common heuristic used when proving practical cryptographic schemes secure.

Several of the CCSDS-recommended cryptographic algorithms are proven secure in the random-oracle model. Hence, this does not add any new assumption to the full protocol. In practice, $\mathcal{H}$ can be implemented preferably with HMAC-SHA2, or KMAC-SHA3.

A variant of this combiner is currently discussed in IETF's CFRG for standardization [OWK23] and is also used in a proposed Internet-Draft for Post-Quantum OpenPGP [KSW22]. Moreover, variants of this design are discussed in CFRG that fix the combined KEMs (c.f., X-Wing [BCD+24]). This has the advantage that the combiner can be optimized for the choices made, but of course comes at the cost of generality.

To highlight this once more: The exact choice of combiner does not matter for the security of the protocol proposed here, as long as the resulting KEM achieves IND-CCA security given that at least one of the combined KEMs does the same.

For the purposes of this report, we will assume the use of the generic combiner [OWK23] with hashed Diffie-Hellman based on X25519 as pre-quantum KEM as it works with all considered schemes. Other curves could of course be used instead, which may or may not change size and/or performance slightly, depending on the replacement.

6.5.5. Signature-Combiners

While designing a secure KEM-combiner is somewhat non-trivial, the opposite is the case for signatures. As long as the standard security notion of Existential Unforgeability under Chosen Message Attacks (EUF-CMA) is the goal, the trivial combiner – double signature – works. For that the message is simply signed with both signature schemes that are used and then the resulting signatures are concatenated for the combined signature. Verification works by splitting the signatures again into two and verifying both of them. If any verification fails the combined signature must be rejected, otherwise the signature is valid.

We remark that this way of combining signatures does not achieve Strong Existential Unforgeability (SEUF-CMA), which requires that an adversary should also be unable to create a different signature for a message for which they already received a valid signature. This is because if one of the signature schemes is broken an attacker may be able to provide a different signature on the message under that scheme while keeping the signature part under the secure scheme, thus providing a new combined signature on the same message. That said, unlike EUF-CMA, SEUF-CMA is not a commonly required property and in particular not needed for any of our proposed protocols.

There do exist more involved security notions for signature combiners (c.f., [BH23]). These are motivated by the use in systems that have to provide legacy options, to achieve backwards compatibility (e.g., can one signature be verified without verifying the other). In the given setting, there is no system for which we have to provide such options and therefore, these notions are not of relevance here.

In places where a pre-quantum signature is necessary, we will assume the use of ECDSA [AN99], in particular with the NIST P-256 curve. ECDSA is listed as acceptable in the CCSDS-document on Cryptographic Algorithms and outperforms the alternatively listed RSA-signatures on almost every metric. As with the KEM-fallback this is largely a representative choice and could be replaced with any other signature-scheme (for example EdDSA [BDL+11]).

6.6. Proposals

In this section we provide outlines for possible protocols with different trade-offs. In general, our proposals can be seen as the main blueprints that can be used (and have been used) to build authenticated key-exchange using KEMs. Thereby they are summarizing the core of all previous proposals for post-quantum secure communication using KEMs and signatures. The difference between the existing previous proposals is mostly motivated by considerations regarding backwards-compatibility or non-cryptographic, protocol-specific considerations.

As a baseline, we consider the traditional approach of using signature key pairs as long-term keys and KEM (traditionally public key encryption) key pairs as ephemeral keys. This approach was for example used in TLS up to version 1.2. Most early proposals for post-quantum secure communication focused on TLS and followed this general approach (see for example [SWZ16, CPS19, PST20]). Next, we discuss different variations of a more recent approach proposed for PQC which only uses KEM key pairs, motivated by the difference in sizes between PQ-KEM and -signatures. This approach was used by the PQ protocols PQWireGuard [HNS+21], KEM-TLS [SSW20] and PQNoise [ADH+22b]. Finally, we discuss some more involved variations of the KEM+Signature approach.

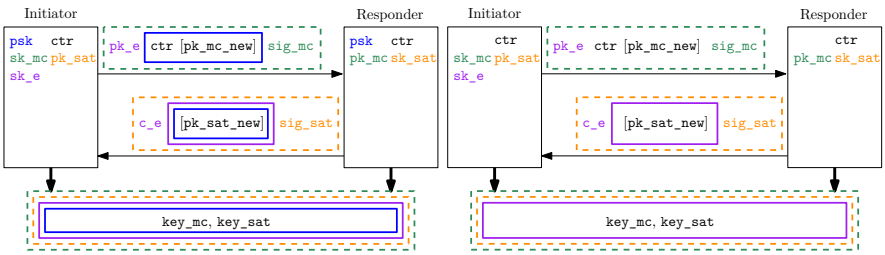

Figure 6.2.: Overview of a KEM+Signature exchange. Solid boxes indicate that confidentiality and in case of the blue boxes authenticity of their content are protected by the same-color secret. Dashed boxes indicate that only the authenticity, but not the confidentiality is protected. Values in square-brackets are optionally updated long-term keys and may be skipped. The left diagram depicts the situation in which there is an uncompromised pre-shared key (psk) and the right one the situation where there is not.

6.6.1. KEM+Signature Exchanges

The traditional approach revolves around using an ephemeral KEM key pair to encapsulate the session key and use a long-term signing key to authenticate the exchange.

This is the general approach that was and is for example used by TLS. In modern versions with Diffie-Hellman key exchange, and before that with RSA.

This kind of exchange can be described by the following Noise-pattern:

```
-> s
<- s
...
-> psk, e, s'[opt1], sig
<- ekem, s'[opt2], sig
-> confirm
```

We provide detailed code of a variant of this proposal in Appendix C.1. Complete code for this proposal would differ from the code shown there in that the AEAD decryption calls require error handling to deal with invalid

messages. KEMs are expected to use implicit rejection and thus always output a key, correctness is then established by using the key in AEAD encryption or decryption.

We note that there may not always be a psk that can be used to derive security from it. In those cases that key should be set to an all-zero bitstring of appropriate length. It will no longer add any security in those cases, but since its inclusion in the handshake is a pure defense-in-depth-measure, this does not significantly endanger the security of the overall protocol. We include the psk because it is computationally cheap to do so and it provides a fairly robust security fallback, not because our analysis relies on it. See Figure 6.2 for a diagram that depicts the difference in protection of the transmitted data depending on whether the used psk is secure.

This proposal is the baseline. It follows a well vetted design for which numerous implementations already exist (without the public key update). Any other proposal would have to beat this one to be considered.

The modification for a partially authenticated version, which derives the authenticity of the responder from outside facts, would differ from this version in that the responder's packet would only include the ciphertext for the ephemeral KEM (`ekem`) and drop the signature and the updated long-term-key.

6.6.2. Dual/Triple-KEM Exchanges

The more recent approach to authentication in a post-quantum setting is the use of a long-term KEM-key to effectively run a challenge-response protocol. If Alice wants to authenticate herself to Bob, she has Bob encapsulate a shared secret for her long-term key and send her the ciphertext. If she is then able to create an AEAD-ciphertext using the shared secret as key, Bob can conclude upon successful decryption that he is talking to Alice. The encapsulated shared secret was only accessible to himself and the owner of the private key (Alice). Since it is (by assumption) not practically possible to create an AEAD-ciphertext for a given key without knowing said key, only Alice or Bob could have created that ciphertext.

Existing examples for PQ protocols that follow this approach are PQWireGuard [HNS+21], KEM-TLS [SSW20] and PQNoise [ADH+22b]. The following proposal is close to the pqKK-pattern of PQNoise up to the optional inclusion of new long-term public keys.

```
-> s
<- s
```

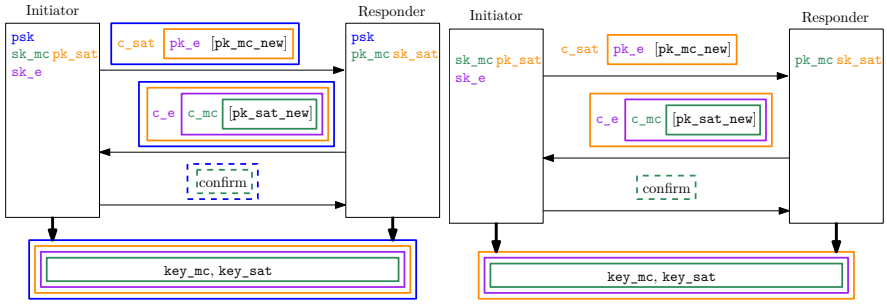

Figure 6.3.: Overview of the Triple-KEM exchange. Solid boxes indicate that the confidentiality of their content is protected by the same-color secret. Solid blue boxes and boxes in the color of the senders's long-term secret furthermore guarantee authenticity at the time at which they are received. Dashed boxes indicate that only the authenticity (when received), but not the confidentiality is protected. By the time the protocol completes (that is when it outputs `key_mc` and `key_sat`) the entire transcript is authenticated. Values in square-brackets are optionally updated long-term keys and may be skipped. The left diagram depicts the situation if there is an uncompromised pre-shared key (psk) and the right one the situation where there is not.

```
...
-> psk, skem, e, s' [opt1]
<- ekem, skem, s'[opt2]
-> confirm
```

The main-differences are that none of the aforementioned protocols have a mechanism to update long-term-keys, that PQNoise directly proceeds to a messaging-phase, whereas our protocol is a pure key-exchange that outputs keys at the end.

This approach has one main advantage over the more traditional signature-based authentication: Post-quantum KEMs tend to be more efficient than signature-schemes, especially in terms of size. Hence, trading signatures for KEMs leads to protocols with lower communication cost. It comes as a nice addition that this approach is secure against replay-attacks by construction.

Just as with the Sign+KEM-approach there may not always be a previous shared key (psk) that can be used to derive security from it. Our response to that problem is the same as with Sign+KEM as well: The psk should in those cases be initialized with an all-zero bitstring, as it is merely used as a defense-in-depth-measure. See Figure 6.3 for a diagram that depicts the difference in protection of the transmitted data depending on whether the used psk is secure.

If only the authenticity of the initiator has to be ensured because the authenticity of the responder can be guaranteed out-of-band, Triple-KEM can be simplified into **Dual-KEM**: Dual-KEM differs from Triple-KEM only in that the initiator would not send a ciphertext for the responder's long-term-key and that the responder's answer would drop the updated long-term-key. The benefit of this approach would lie in a more compact first packet and less computation because a third of the asymmetric operations could be skipped. The disadvantage is that Dual-KEM does not itself guarantee any authenticity of the responder. Sections 6.9 and 6.10 provide detailed descriptions of Triple-KEM and Dual-KEM.

6.6.3. Variants

Below we discuss two variations of the Sign+KEM-approach using stateful (hash-based) signatures. First, we discuss what happens if one simply replaces the signature scheme with a stateful one. Then we discuss how this approach can be optimized using one-time signature chains.

Stateful Signatures

The above two proposals work for any standard KEM and signature scheme. We remark that if the number of uses of a long-term key is limited by policy, stateful signatures like XMSS or LMS could potentially be used. The big downside of stateful signatures is that their secret key changes over time. If an old key state is reused, they become insecure. This can happen for example as a result of using the secret key of an outdated backup. While this forms an unacceptable hazard for systems that face end-users, keeping a state may be a manageable task in an expert-controlled environment like a mission-control center or a satellite similar to how stateful hash-based signatures are used in code signing.

Furthermore, using LMS or XMSS as (stateful) signature-schemes has also advantages:

- Both have great performance- and size-characteristics.

- Unlike the NIST-winners they are already standardized.
- The schemes do not introduce any new security assumptions but rely on the security of hash functions – a necessity for any signature scheme that can handle arbitrary length messages.
- For that reason, ANSSI states that there is no need for a hybrid solution when using hash-based signatures.

The resulting protocol is identical to the KEM+Signature-proposal, except that the secret keys for the signature-scheme change after each signature and that occasional updates of the long-term keys are no longer optional.

Signature Chains

If a satellite only communicates with one mission-control center (or multiple, but in such a way that the interactions are completely independent of each other), it is possible to go even further and move to signature-chains. This idea was proposed in the context of sensor networks where computational resources are sparse. In the vanilla version of this idea, each key pair is used to authenticate one message, together with the public key of the next key pair. In this case, a so-called one-time signature scheme like WOTS can be used, which forms the main ingredient of stateful hash-based signature schemes, including XMSS and LMS.

As a Noise-pattern this would look as follows:

```
-> s
<- s
...
-> psk, e, s', sig
<- ekem, s', sig
```

This solution can be an attractive solution due to the reduced computational cost and the comparatively small signature size. On the downside, this involves more complicated key management as the “long-term” key pair of a party has to be updated with every key exchange.

Care has to be taken to handle exceptional cases like if a message got lost, or if either side got interrupted during the exchange. On both sides the countermeasure is the same. All randomness required by a party to run the protocol is sampled as first step of the protocol and stored in non-volatile memory. Only then do the parties run the protocol, using the stored

randomness. This way the protocol becomes deterministic, and an aborted protocol run can be reproduced with exactly the same messages. As long as parties only update the locally stored key pair in storage when the protocol successfully terminated, de-synchronization can be avoided.

6.6.4. Generic Comparison

As outlined in Section 6.7.1, the Triple-KEM approach has the potential advantage, that it allows to mix and match different KEMs to protect different properties using different trade-offs. If long-term-keys are for example never updated, it would be possible to instantiate them with Classic McEliece to benefit from the small ciphertexts but use Kyber for ephemeral keys where the public key has to be transmitted on every exchange. This would improve the performance, the bandwidth requirements and potentially even the security.

An advantage of the signature-based approach on the other hand is that it technically allows to design a protocol that saves half a round-trip because it does not strictly require a key-confirmation message. Assuming that a sufficiently secure replay-protection is in place, the first message in the protocol can be assumed to be authentic by the satellite, because it is by assumption not possible for an attacker to forge the included signature and the replay-protection prevents replays. Similarly, the response could be assumed to be authentic by the receiver for the same reasons.

The problem with this approach is that it is very fragile and minor issues that would be fine with a key-confirmation message may cause persistent denial-of-service attacks:

Any weakness in the replay-protection-system would not just allow for the initiation of a handshake via replaying the previous message, but would in fact complete the protocol, possibly replacing a critical communication-key on the satellite with one that is unknowable to mission-control if mission-control already discarded its ephemeral keys. Replay-protection systems are however not trivial to get right, typical approaches include counters which requires maintaining state or time-based approaches which require synchronized clocks *and* a very tight control of the acceptance-window of a message: If the window is open for even slightly too long, replays become possible for attackers who have very low-latency access to previously sent messages. A window that is slightly too short will prevent that attack but may also prevent the honest user from sending a message within that window. In the context of satellites that may change their distance from at least the ground-stations very quickly and may in the extreme orbit bodies other than earth, an exact calculation

of that window, while certainly possible, is sufficiently non-trivial to make its inclusion in a cryptographic module questionable.

Even if there are no problems with the replay-protection-system, a further potential issue could occur if the satellite's response is lost for whatever reason. In this case the satellite would perform the requested key-update, but the ciphertext needed by mission-control to compute the shared secret would never arrive, causing functionally the same problem described in the previous case. We remark that while transport-layer protocols could mitigate this issue to an extent, they would functionally turn the approach into a more-than-one round-trip protocol.

For these reasons we can already state here, that we do not recommend versions of the signature-based approaches that skip key-confirmation.

6.7. Alternative approaches not explored

Here we discuss concepts we briefly considered and decided against. The first one is a different approach towards combining PQC and ECC. The second proposal is an alternative way to achieve forward-secrecy.

6.7.1. Hybrid scheme

The common way to combine PQC and ECC is via the use of primitive combiners. This approach has the advantage that it massively simplifies the analysis of the greater protocol, because the protocol-analysis will generally not take the concrete instantiation of a primitive into consideration, but merely that the primitive is secure. The disadvantage is that the sizes and computation of the combined schemes add up and that some extra calls to a key-derivation function are needed.

Alternatively, it is also possible to use different instantiations of the same primitive in the actual protocol. For example, we can instantiate the long-term KEM key pair in the triple-KEM proposal with Classic McEliece, and the ephemeral key pair with Kyber. This approach was introduced by PQWireGuard as a means to optimize package size (though there a variant of Saber is used for the ephemeral key instead of Kyber and the scheme recommends using an ECC-fallback regardless). This exploits that McEliece has extremely small ciphertexts and if the public key never has to be sent in the protocol, its size does not matter too much.

This approach achieves full hybrid security in the absence of key compromise. However, as soon as a key (long-term or short-term) is compromised,

security is fully based on the security of the scheme with the non-compromised key. Moreover, authentication becomes non-hybrid in this case. Because of these issues we did not further consider this option as a full replacement of the regular hybrid approach. We do note however that it can still act as a defense-in-depth-measure and may be useful for different reasons (such as performance- or bandwidth-advantages).

Table 6.3.: Sizes for a Triple-KEM Kyber-McEliece Hybrid in bytes, without and with long-term key update. Here McEliece is used for long-term keys, Kyber for ephemeral keys

Scheme	Packet 1	Packet 2	Packet 3
TK(Kyber512,McEliece348864)	928	896	16
TKU(Kyber512,McEliece348864)	262048	262032	16

6.7.2. Forward Secrecy through Symmetric Ratchet

Forward Secrecy states that communication has to remain confidential even if an involved party becomes compromised at a later point in time. The traditional way to achieve this is through the use of ephemeral key-exchanges as part of a handshake – a later compromise of the long-term secrets does not allow to break confidentiality of previous communication once the ephemeral secrets of that communication are deleted.

We note that this is not the only way to achieve this property though: It is also possible to use a traditional "symmetric ratchet". The core-idea of that approach is that instead of updating the shared secret via a new key-exchange, new keys can essentially be derived from old keys via a one-way function. This way all involved parties can perform key updates on their own without having to interact with their peers, as long as everyone agrees on the update policy. This approach is remarkably efficient and is often worth doing, but while it protects against future compromise it cannot recover from past compromise.

The approach made it into our final version in so far as that we mix the previous key into the hash-object as a defense-in-depth-measure, but the inability to create new and independent key-material meant that we did not rely on it as a primary defense.

6.8. Evaluation

The packet-sizes for the various possible protocols are depicted in Table 6.4.

The hybrid-versions assume that a KEM-combiner is used with a KEM-version of X25519, but without turning WOTS+ into a hybrid. These numbers would change slightly depending on whether only EKEM should be hybrid or whether the signatures should also be made hybrid. That said, the comparison clearly demonstrates that the use of a hybrid elliptic curve scheme comes at little cost with regards to packet-size compared to using only the post-quantum scheme. We do not consider instantiating ephemeral KEMs with Classic McEliece due to the large size of the public key.

All of these sizes assume the use of an AEAD-scheme that adds a constant 16 bytes to the length of the plaintext; this assumption holds for many widely used schemes, in particular AES-GCM with a full-sized authentication-tag. The sizes are however independent of the choices made for the involved hash-functions (except for the case of hash-based signatures), as the protocol does not transmit any hashes.

We note that all of these proposals offer forward-secrecy, post-compromise security, updateable long-term keys and some form of hybrid security. This also holds if the network fragments packets, but in the event that fragmentation causes data to arrive out-of order or not at all, the protocol as presented would not be able to reorder or re-request them, and instead assume an attack and drop the connection. The layer handling fragmentation would be required to ensure the correct ordering and re-requesting of lost packets if dropped connections are an issue.

6.8.1. Conclusion

Our primary recommendation is Triple-KEM(Kyber+X25519):

- Instead of two PQC and ECC schemes in case of Sign+KEM, only one scheme each has to be implemented, reducing the chance of mistakes and the attack surface.
- Packets are generally smaller than for Sign+KEM, except for Kyber + Falcon without key update. However, Falcon amplifies the above issue as it is known to be notoriously hard to implement securely due to the use of floating-point arithmetic.
- In terms of speed, we expect Triple-KEM(Kyber+X25519) to clearly outperform Sign+KEM(Kyber512+X25519+Falcon+ECDSA).

Table 6.4.: Sizes of different instantiations in bytes. SK = Sign + KEM, SKU = Sign+KEM with update of long-term key, SC = Signature-Chain, TK = Triple-KEM, TKU = Triple-KEM with update of long-term key.

Scheme	Packet 1	Packet 2	Packet 3	Security-Level
SK(Kyber512+X25519+Dilithium+ECDSA)	3348	3300	16	1
SKU(Kyber512+X25519+Dilithium+ECDSA)	4692	4644	16	1
SK(Kyber512+X25519+Falcon+ECDSA)	1594	1546	16	1
SKU(Kyber512+X25519+Falcon+ECDSA)	2523	2475	16	1
SK(Kyber512+X25519+XMSS-SHA2_10_256)	3364	3316	16	1
SKU(Kyber512+X25519+XMSS-SHA2_10_256)	3428	3380	16	1
SC(Kyber512+X25519,WOTS+(32,16))	3024	2992	16	1
SC(Kyber768+X25519,WOTS+(32,16))	2408	3312	16	3
SC(Kyber1024+X25519,WOTS+(32,16))	3792	3792	16	5
SC(Kyber1024+X25519,WOTS+(64,16))	10032	10032	16	5
TK(Kyber512+X25519)	1664	1632	16	1
TKU(Kyber512+X25519)	2496	2480	16	1
TK(Kyber768+X25519)	2368	2272	16	3
TKU(Kyber768+X25519)	3584	3504	16	3
TK(Kyber1024+X25519)	3232	3232	16	5
TKU(Kyber1024+X25519)	4832	4848	16	5
TK(Kyber512,McEliece348864)	928	896	16	1
TKU(Kyber512,McEliece348864)	262048	262032	16	1

This comes with the variant Dual-KEM(Kyber+X25519):

- Besides giving up responder-authenticity on the protocol-level, the variants are very similar.
- The bandwidth-savings equate to the size one Kyber-ciphertext in all interactions; additionally, not updating the long-term-key saves the size of Kyber-public-key.
- Dual-KEM can clearly be expected to have better performance than Triple-KEM, both for computation time and (more important) bandwidth.

Our second choice is still Sign+KEM(Kyber512+X25519+Falcon+ECDSA).

- The sizes are best in class as long as no key update is done and even then they are at a close second place.
- The scheme still outperforms the hash-based proposals even though it adds ECC.
- It avoids the necessity to maintain a state.
- However great care needs to be taken to implement Falcon securely.

The proposals using XMSS / LMS, or one-time signature chains may be of interest in case that a mission already has an implementation of the primitive available on the device. In this case, savings in code-size could be achieved and state-handling is likely already supported.

6.9. Triple-KEM in Detail

The following listing provides a detailed description of Triple-KEM. It uses the following primitives as building-block:

- `NoiseHashObject`, a construction that Noise-style protocols use and that was first formalized as its own primitive in PQNoise [ADH+22b]. For a complete description see Section 6.11.2; intuitively this primitive consists of a state (here: `prho_state`) that can be used with the two functions `input` and `finalize`: Input takes a state and a bitstring as arguments and returns an updated state and a freely choosable number of outputs. If the bitstring that has to be provided as argument is

pseudo-random, then all subsequent outputs are pseudo-random as well. `finalize` works in the same way, except that it does not output an updated state.

For simplicity we use class-style notation with the hash-object, that is we write `out = state.input(in)` instead of `state, out = input(state, in)`.

- `H`, a collision-resistant hash-function, for example SHA3. Functionally we use this to implement a hash-object that uniquely hashes the network-transcript up to a certain point. Multiple values are always added seperately (`H(H(state, x), y)` instead of `H(state, x, y)`) to ensure domain-separation.

- The value `PROTOCOL_TOKEN` is a token unique to the protocol; it can prevent protocol confusion and provides a degree of domain-separation. This parameter is public and its concrete value freely choosable. Possible values would for example be the ASCII-encoding of "SDLSP-KeyReplacment-3KEM-3Kyber-Version1", UUIDs or values unique to the network that the satellite is supposed to participate in.

- `IKEM`, `RKEM` The long-term KEMs used by the initiating party (usually: mission-control) and the responding party (usually: the satellite). In our recommendation both of these are set to Kyber+X25519.

- `EKEM` The ephemeral KEM, in our recommendation Kyber+X25519

- `AEAD` An **A**uthenticated (symmetric) **E**ncryption-scheme with support for **A**ssociated **D**ata, for example AES-GCM. The inputs to `AEAD.enc` are the key, the plaintext, a nonce, and the associated data; the output is the ciphertext. The inputs to `AEAD.dec` are the key, the ciphertext, the nonce, and the associated data; the output is the plaintext or failure (if authentication fails).

- `KDF` a Key-Derivation Function, for example HMAC-SHA2

For KEM `KEM`, `KEM.enc` produces the ciphertext and the shared key using the receiver's public key; `KEM.dec` produces the shared key from the ciphertext using the receiver's secret key.

The following protocol does not include statements for error handling. If the AEAD returns failure this error must be caught and handled appropriately. Benign reasons may be decryption failures in the KEMs leading to

mismatching secret keys or communication failures introducing errors. The latter should be caught by error-correcting codes.

The final output of the protocol consists of the two keys `key_mc` and `key_sat` that can then be used to encrypt data symmetrically (primarily via SDLSP). We output two keys in order to enable assigning one key as exclusive sending-key to each involved party as a robustness measure (in particular there is less danger of nonce-reuse if every party has its own sending-key) but note that both keys are equally secure and that only using one of them is possible.

Additionally, `h_6` forms a computationally unique (based on the collision-resistance of `H`) handshake-hash that can safely be used as a session-identifier to uniquely identify a session. The final algorithms on both sides `send_3` and `receive_3` can be altered to additionally output the value of `h_6` without any security impact if such a session identifier is required on a higher protocol level.

(The following notation largely aligns with Python; "+" when used with respect to byte-sequences means concatenation.)

```
def send_1(update_longterm: bool):
    prho_state_mc = NoiseHashObject.create()
    psk = (key_mc + key_sat) if key_mc is not None else b'\0\0...'
    h_0 = H(PROTOCOL_TOKEN)
    k_0 = prho_state_mc.input(psk)
    c_sat, k_sat = RKEM.enc(pk_sat)
    c_0 = AEAD.enc(k_0, c_sat, 0, h_0)
    h_1 = H(h_0, c_0)
    k_1 = prho_state_mc.input(k_sat)
    pk_e, sk_e = EKEM.gen()
    payload = pk_e
    if update_longterm:
        pk_mc_new, sk_mc_new = IKEM.gen()
        payload += pk_mc_new
    c_1 = AEAD.enc(k_1, payload, 0, h_1)
    h_2 = H(h_1, c_1)
    send(c_0, c_1)

def receive_1((c_0, c_1)):
    prho_state_sat = NoiseHashObject.create()
    psk = (key_mc + key_sat) if key_mc is not None else b'\0\0...'
    h_0 = H(PROTOCOL_TOKEN)
    k_0 = prho_state_sat.input(psk)
    c_sat = AEAD.dec(k_0, c_0, 0, h_0)
    k_sat  = RKEM.dec(sk_sat, c_sat)
    h_1 = H(h_0, c_0)
    k_1 = prho_state_sat.input(k_sat)
    pk_e, pk_mc_new = AEAD.dec(k_1, c_1, 0, h_1)
```

```
    h_2 = H(h_1, c_1)

def send_2(update_longterm: bool):
    c_e, k_e = EKEM.enc(pk_e)
    c_2 = AEAD.enc(k_1, c_e, 1, h_2)
    h_3 = H(h_2, c_2)
    k_2 = prho_state_sat.input(k_e)
    c_mc, k_mc = IKEM.enc(pk_mc)
    c_3 = AEAD.enc(k_2, c_mc, 0, h_3)
    h_4 = H(h_3, c_3)
    k_3 = prho_state_sat.input(k_mc)
    if update_longterm:
        # This does not enforce knowledge of the new key,
        # but that should not be a problem:
        pk_sat_new, sk_sat_new = RKEM.gen()
        c_4 = AEAD.enc(k_3, pk_sat_new, 0, h_4)
        h_5 = H(h_4, c_4)
        send(c_2, c_3, c_4)
    else:
        h_5 = h_4
        send(c_2, c_3)

def receive_2((c_2, c_3, c_4)):
    c_e = AEAD.dec(k_1, c_2, 1, h_2)
    k_e = EKEM.dec(sk_e,c_e)
    h_3 = H(h_2, c_2)
    k_2 = prho_state_mc.input(k_e)
    c_mc = AEAD.dec(k_2, c_3, 0, h_3)
    k_mc = IKEM.dec(sk_mc, c_mc)
    h_4 = H(h_3, c_3)
    k_3 = prho_state_mc.input(k_mc)
    if c_4 != "":
        pk_sat_new = AEAD.dec(k_3, c_4, 0, h_4)
        h_5 = H(h_4, c_4)
    else:
        h_5 = h_4

def send_3():
    c_5 = AEAD.enc(k_3, "", 1, h_5)
    send(c_5)
    if pk_sat_new:
        pk_sat = pk_sat_new
        pk_sat_new = None
    if sk_mc_new:
        sk_mc = sk_mc_new
        sk_mc_new = None
    h_6 = H(h_5, c_5)
    pre_key_mc,pre_key_sat = prho_state_mc.finalize("")
    key_mc = KDF(pre_key_mc, h_6)
    key_sat = KDF(pre_key_sat, h_6)
```

```
79      return key_mc, key_sat
80
81  def receive_3(c_5):
82      m = AEAD.dec(k_3, c_5, 1, h_5)
83      if m != "":
84          error()
85      if sk_sat_new:
86          sk_sat = sk_sat_new
87          sk_sat_new = None
88      if pk_mc_new:
89          pk_mc = pk_mc_new
90          pk_mc_new = None
91      h_6 = H(h_5, c_5)
92      pre_key_mc,pre_key_sat = prho_state_sat.finalize("")
93      key_mc = KDF(pre_key_mc, h_6)
94      key_sat = KDF(pre_key_sat, h_6)
95      return key_mc, key_sat
```

6.10. Dual-KEM in Detail

The easiest way to turn the above triple-KEM-protocol into a dual-KEM-protocol would be to instantiate the satellite-KEM with a Null-KEM, that uses the empty string as secret-key, public-key, ciphertext and shared secret and just returns those values from gen, enc and dec. As an added bonus this even preserves the validity of all proofs that do not rely on any properties of the satellite-KEM, such as the below proofs for initiator-authenticity and confidentiality.

The downside of this approach is however that it results in a protocol that does slightly more work and sends slightly more data than necessary. For this reason, we provide a dedicated Dual-KEM, that has everything related to the satellite-KEM fully stripped out.

The following listing provides a detailed description of Dual-KEM. It uses the same building-blocks as before and closely follows the design of Triple-KEM. It skips the use of some indices in the variables (there is for example no `h_1`) to stay consistent with the naming in Triple-KEM.

```
1  def send_1(update_longterm: bool):
2      prho_state_mc = NoiseHashObject.create()
3      psk = (key_mc + key_sat) if key_mc is not None else b'\0\0...'
4      h_0 = H(PROTOCOL_TOKEN)
5      k_0 = prho_state_mc.input(psk)
6      pk_e, sk_e = EKEM.gen()
7      payload = pk_e
8      if update_longterm:
```

```
        pk_mc_new, sk_mc_new = IKEM.gen()
        payload += pk_mc_new
    c_1 = AEAD.enc(k_0, payload, 0, h_0)
    h_2 = H(h_0, c_1)
    send(c_1)

def receive_1(c_1):
    prho_state_sat = NoiseHashObject.create()
    psk = (key_mc + key_sat) if key_mc is not None else b'\0\0...'
    h_0 = H(PROTOCOL_TOKEN)
    k_0 = prho_state_sat.input(psk)
    pk_e, pk_mc_new = AEAD.dec(k_0, c_1, 0, h_0)
    h_2 = H(h_0, c_1)

def send_2():
    c_e, k_e = EKEM.enc(pk_e)
    c_2 = AEAD.enc(k_0, c_e, 1, h_2)
    h_3 = H(h_2, c_2)
    k_2 = prho_state_sat.input(k_e)
    c_mc, k_mc = IKEM.enc(pk_mc)
    c_3 = AEAD.enc(k_2, c_mc, 0, h_3)
    h_5 = H(h_3, c_3)
    k_3 = prho_state_sat.input(k_mc)
    send(c_2, c_3)

def receive_2((c_2, c_3)):
    c_e = AEAD.dec(k_0, c_2, 1, h_2)
    k_e = EKEM.dec(sk_e,c_e)
    h_3 = H(h_2, c_2)
    k_2 = prho_state_mc.input(k_e)
    c_mc = AEAD.dec(k_2, c_3, 0, h_3)
    k_mc = IKEM.dec(sk_mc, c_mc)
    h_5 = H(h_3, c_3)
    k_3 = prho_state_mc.input(k_mc)

def send_3():
    c_5 = AEAD.enc(k_3, "", 1, h_5)
    send(c_5)
    if sk_mc_new:
        sk_mc = sk_mc_new
        sk_mc_new = None
    h_6 = H(h_5, c_5)
    pre_key_mc,pre_key_sat = prho_state_mc.finalize("")
    key_mc = KDF(pre_key_mc, h_6)
    key_sat = KDF(pre_key_sat, h_6)
    return key_mc, key_sat

def receive_3(c_5):
    m = AEAD.dec(k_3, c_5, 1, h_5)
    if m != "":
```

```
59          error()
60      if pk_mc_new:
61          pk_mc = pk_mc_new
62          pk_mc_new = None
63      h_6 = H(h_5, c_5)
64      pre_key_mc,pre_key_sat = prho_state_sat.finalize("")
65      key_mc = KDF(pre_key_mc, h_6)
66      key_sat = KDF(pre_key_sat, h_6)
67      return key_mc, key_sat
```

6.11. Analysis

The Triple-KEM-protocol provides the following properties:

1. **Confidentiality**: Honestly generated keys remain indistinguishable from randomness (=confidential) if the ephemeral randomness used during their creation remains confidential.

2. **Authenticity**: A party cannot be impersonated, as long as its long-term public key and the peer's ephemeral randomness remain uncompromised.

3. Honestly generated keys remain confidential if the pre-shared key remains uncompromised.

4. Honestly generated keys remain confidential as long as one party's long-term key and the peer's ephemeral randomness remain uncompromised.

5. As long as a connection remains confidential (see above), no passive attacker can learn more about a new long-term public-key than can be extracted from ciphertexts for that public key.

We will prove properties 1 and 2, as they cover the requirements. Properties 3 and 4 fall into the category of defense-in-depth where they are useful to mitigate some attacks that may result from cryptanalytic progress or issues with the implementation but should not be relied upon on their own and are not required to meet the requirements. The same holds for Property 5, which given the lack of a need for privacy is not itself relevant here but may mitigate some attacks by denying the adversary knowledge about targeted public-keys.

The Dual-KEM-protocol provides most of the same properties, except for the authenticity of the responder:

1. **Confidentiality**: Honestly generated keys remain indistinguishable from randomness (=confidential) if the ephemeral randomness used during their creation remains confidential.

2. **Authenticity**: The initiator cannot be impersonated, as long as its long-term public key and the responder's ephemeral randomness remain uncompromised.

3. Honestly generated keys remain confidential if the pre-shared key remains uncompromised.

4. Honestly generated keys remain confidential as long as one party's long-term key and the peer's ephemeral randomness remain uncompromised.

5. As long as a connection remains confidential (see above), no passive attacker can learn more about a new long-term public-key than can be extracted from ciphertexts for that public key.

Considering the similarity between the protocols, the proofs for the Triple-KEM are, with the exception of the responder-authenticity, fully applicable. Given the lack of built-in responder-authenticity, Dual-KEM only achieves our targeted security-notion if that authenticity is provided from other sources. Considering that the motivation behind not requiring that built-in authenticity is that impersonating an orbiting satellite on a physically narrow channel is assumed to be infeasible, we will make that assumption for the proof.

We note that additionally (although out of scope for this work), the use of a pre-shared secret allows to prevent a denial-of-service attack based on the cryptographic protocol. More specifically, the use of the pre-shared secret allows the responder, i.e., the satellite, to reject connection requests based on the first message and thereby prevents an attacker from filling the memory with invalid partial sessions.

6.11.1. Model

We will use a modified version of the BR-model [BR94] that we will however describe relative to the nowadays more common eCK-model [LLM07]. At their core all three of these models works in a setting where n_p parties can run up to n_s key-exchanges (or „sessions“) each between them to compute shared secrets. The adversary is in full control of the network and can tell parties when to send or receive network-packets. Eventually the adversary gets to choose a target-session for which it receives a candidate for the

shared secret, that depending on the challenge-bit is either the actual shared secret or a random and independent value of the same distribution. If the adversary manages to guess the challenge-bit correctly while observing certain freshness-conditions, it wins. A protocol is considered secure if there is no Quantum Polynomial Time (QPT) attacker whose win-probability is non-negligibly higher than 0.5.

Our biggest modification to the eCK-model is that we do not provide an oracle to reveal the ephemeral key to outlaw some classes of attacks that are out-of-scope, which is in line with the BR-model. While this results in a strictly weaker security-notion, we still benefit from the familiarity of eCK and note that ephemeral key corruptions in the real world are almost exclusively caused by bad random-number-generators and that the best way to prevent them is to harden the random-number-generator.

This simplifies the freshness-predicates, because several failure-conditions depend on the adversary querying ephemeral keys. Specifically, a session sid is now considered fresh, unless:

- The adversary queried to reveal the session key of sid or of its peered session (if it exists).
- No peered session to sid exists and the adversary queried the long-term key of the peer before the completion of the session.

The second, much more minor modification is that our version generates an initiator- and a responder-key per session pair, rather than a single secret. As acquiring either of them is considered a break, this is largely a technicality.

Furthermore we introduce the variables n_i and n_r that refer to the maximum number of initiators and responders respectively: While both of these values are upper-bounded by n_p, especially n_i will most of the time be significantly smaller in our use-case, allowing us to get a better security-statement at no cost. For the same reason we separate n_s into n_{s_i} and n_{s_r} as the maximum number of sessions that an initiator or responder may respectively run: For the same reason for which there are often few initiators, these few initiators may run a significantly higher number of sessions than most responders. Lastly, we remark that post-quantum KEMs often do not offer perfect correctness, that is even an honestly generated ciphertext might not successfully decapsulate to the same shared secret that the sender obtained when encapsulating. We will denote the probability of this happening to a KEM KEM as $\mathsf{KEM}.\delta$.

We will henceforth refer to this notion as "BR'", to indicate that it is a modified BR-model.

6.11.2. Pseudo-Random Hash-Object (PRHO)

A major component of our protocol is the precise mechanism used for hashing. PQNoise [ADH+22b] introduced an abstracted interpretation of these hash-chains in the form of a (Noise-) Pseudo-Random Hash-Object (PRHO), that produces an unlimited number of values that are indistinguishable from randomness, once it has been fed randomness. It furthermore proved that the way the hashing is done securely implements such a PRHO. Because this instantiation is precisely what we need as well, we point to the PQNoise-paper for a more detailed discussion and analysis and will henceforth treat the protocol as using a Noise-PRHO directly.

Formally a Noise-PRHO is a tuple of three deterministic algorithms: create, input, finalize, and an integer-constant n.

create$(1^\lambda) \to s$ (conceptually) takes a security-parameter λ and returns a state s.

input$(s, m) \to s', h$ takes a state s and message $m \in \mathbb{B}^*$ and returns a new state s' and a list $h \in (\mathbb{B}^\lambda)^n$ of hashes of length n .

finalize$(s, m) \to h$ works like input, except that it does not return a state.

For $n = 2$, which is the relevant case for us, the instantiation of this primitive that Noise and its derivatives use and that we recommend as well then looks as follows:

```
def create():
    return ""

def input(state, m):
    tmp = hmac_hash(state, m)
    new_state = hmac_hash(tmp, b"\x00")
    h1 = hmac_hash(tmp, new_state + b"\x01")
    h2 = hmac_hash(tmp, h1 + b"\x02")
    return new_state, [h1, h2]

def finalize(state, m):
    h_0, [h_1, h_2] = input(state, m)
    return [h_0, h_1]
```

We remark that the only situation where we require both outputs are the ones where finalize is called, making it a desirable optimization to skip the computation of `h_2` in all cases; we only include it here for the sake of staying in line with pre-existing literature.

6.11.3. Outline

While BR' covers both authenticity and confidentiality at the same time, we treat them largely separately as we consider that approach more intuitive.

Authenticity

The way the protocol provides authenticity is identical for both parties: It uses a long-term KEM to derive a shared secret that should only be known to the encapsulating party and the owner of the secret key and only accepts once the encapsulating party receives a response that is encrypted with an AEAD-scheme that uses a key derived from that shared secret. On a high-level they both work as follows:

- In the first game-hop we abort if there is ever a hash-collision. This is sound, because this should never happen if the hash-function is secure.
- In the second game-hop we guess the impersonated party and its peer and the attacked session and abort upon a wrong guess.
- In the third game-hop we replace the shared secret with a random string. We derive the security of this step from the security of the long-term KEM in question.
- In the fourth game-hop we replace the subsequent outputs of the PRHO with randomness. We derive the security of this step from the security of the hash-object.
- In the fifth game-hop we abort the interaction upon receiving a reply in the challenge-session. This is secure, because a valid reply could be used to break the AEAD-security.
- At this point it is trivially impossible for the adversary to impersonate a party in the challenge-session.

Confidentiality

Once authenticity is fully established, confidentiality only has to deal with passive attacks. As we do not use keys that result from sessions that did not reach the stage where the final key was accepted, and because any session that reached that stage without being fully honest would be a break of authenticity, the only sessions that remain to be considered are sessions in which all messages are delivered honestly between the parties.

- In the first game-hop we abort if there is ever a hash-collision. This is sound, because this should never happen if the hash-function is secure.
- In the second game-hop we guess the attacked initiator and initiator-session and abort upon a wrong guess.
- In the third game we abort if there is ever a second honest initiator-session that uses the same ephemeral public key as the target-session.
- In the fourth game we guess the peered responder and responder-session that is targeted and abort upon a wrong guess.
- In the fifth game we abort if there is ever an honest responder session that recreates the ciphertext for the ephemeral KEM in a non-targeted session.
- In the sixth game we abort if a non-partnered session shares the handshake-hash with the target-session. (This is purely conceptual at this point.)
- In the seventh game-hop we replace the shared secret with a random string. We derive the security of this step from the security of the ephemeral KEM in question.
- In the eighth game-hop we replace the subsequent outputs of the PRHO with randomness. We derive the security of this step from the security of the hash-object.
- In the ninth game-hop we replace the returned keys with randomness. We derive the security of this step from the PRF-security of the used key-derivation-function.
- At this point the final keys are random, which restricts the adversary to guessing a random bit with advantage 0.

6.11.4. Formal Security-Statements

Theorem 23. *There is no polynomial-time quantum-adversary that can win the BR'-game against Triple-KEM, with more than negligible probability. Specifically, we find for all such adversaries $\mathcal{A}$:*

$$\mathsf{Adv}^{BR'}_{\mathcal{A},\mathsf{3KEM}}(1^\lambda) \leq \left(\begin{array}{rcl} & 3 & \cdot \ \mathsf{Adv}^{coll\text{-}res}_{\mathcal{A}_1,\mathsf{H}}(1^\lambda) \\ + & n_i \cdot n_{s_i} & \cdot \ \mathsf{EKEM}.\delta \\ + & n_i \cdot n_{s_i} \cdot n_r \cdot n_{s_r} \cdot 3 & \cdot \ \mathsf{Adv}^{IND\text{-}CCA}_{\mathcal{A},\mathsf{EKEM}}(1^\lambda) \\ + & n_{s_r} \cdot n_i \cdot n_r \cdot \frac{1}{1-\mathsf{IKEM}\cdot\delta} & \cdot \ \mathsf{Adv}^{IND\text{-}CCA}_{\mathcal{A}_4,\mathsf{IKEM}}(1^\lambda) \\ + & n_{s_i} \cdot n_i \cdot n_r \cdot \frac{1}{1-\mathsf{RKEM}\cdot\delta} & \cdot \ \mathsf{Adv}^{IND\text{-}CCA}_{\mathcal{A}_4,\mathsf{RKEM}}(1^\lambda) \\ + & n_i \cdot n_{s_i} \cdot n_r \cdot n_{s_r} \cdot 3 & \cdot \ \mathsf{Adv}^{PRHO}_{\mathcal{A},\mathsf{NHO}}(1^\lambda) \\ + & (n_{s_i} + n_{s_r}) \cdot n_i \cdot n_r & \cdot \ \mathsf{Adv}^{EUF\text{-}CMA}_{\mathcal{A}_6,\mathsf{AEAD}}(1^\lambda) \\ + & n_i \cdot n_{s_i} \cdot n_r \cdot n_{s_r} \cdot 2 & \cdot \ \mathsf{Adv}^{PRF}_{\mathcal{A},\mathsf{KDF}}(1^\lambda) \end{array}\right)$$

Proof. This follows directly from adding up the losses in Theorems 25-27, noting that $n_i \cdot n_{s_i} \cdot n_r + n_i \cdot n_r \cdot n_{s_r} + n_i \cdot n_{s_i} \cdot n_r \cdot n_{s_r} \leq n_i \cdot n_{s_i} \cdot n_r \cdot n_{s_r} \cdot 3$, when summarizing the PRHO-losses and simplifying the result. □

Theorem 24. *There is no polynomial-time quantum-adversary that can win the BR'-game against Dual-KEM, with more than negligible probability. Specifically, we find for all such adversaries $\mathcal{A}$:*

$$\mathsf{Adv}^{BR'}_{\mathcal{A},\mathsf{2KEM}}(1^\lambda) \leq \left(\begin{array}{rcl} & 2 & \cdot \ \mathsf{Adv}^{coll\text{-}res}_{\mathcal{A}_1,\mathsf{H}}(1^\lambda) \\ + & n_i \cdot n_{s_i} & \cdot \ \mathsf{EKEM}.\delta \\ + & n_i \cdot n_{s_i} \cdot n_r \cdot n_{s_r} \cdot 3 & \cdot \ \mathsf{Adv}^{IND\text{-}CCA}_{\mathcal{A},\mathsf{EKEM}}(1^\lambda) \\ + & n_{s_r} \cdot n_i \cdot n_r \cdot \frac{1}{1-\mathsf{IKEM}\cdot\delta} & \cdot \ \mathsf{Adv}^{IND\text{-}CCA}_{\mathcal{A}_4,\mathsf{IKEM}}(1^\lambda) \\ + & n_i \cdot n_{s_i} \cdot n_r \cdot n_{s_r} \cdot 2 & \cdot \ \mathsf{Adv}^{PRHO}_{\mathcal{A},\mathsf{NHO}}(1^\lambda) \\ + & n_{s_r} \cdot n_i \cdot n_r & \cdot \ \mathsf{Adv}^{EUF\text{-}CMA}_{\mathcal{A}_6,\mathsf{AEAD}}(1^\lambda) \\ + & n_i \cdot n_{s_i} \cdot n_r \cdot n_{s_r} \cdot 2 & \cdot \ \mathsf{Adv}^{PRF}_{\mathcal{A},\mathsf{KDF}}(1^\lambda) \\ + & & \mathsf{Adv}^{BR'_{Case\ A}}_{\mathcal{A},\mathsf{2KEM}}(1^\lambda) \end{array}\right),$$

where $\mathsf{Adv}^{BR'_{Case\ A}}_{\mathcal{A},\mathsf{2KEM}}(1^\lambda)$ refers to the maximum achievable advantage for the adversary to cause an unpeered, complete initiator-session.

Proof. This follows directly from adding up the losses in Theorems 26 and 27 as well as $\mathsf{Adv}^{\mathsf{BR'}_{\mathrm{Case\ A}}}_{\mathcal{A},\mathsf{2KEM}}(1^\lambda)$ and, noting that $n_i \cdot n_r \cdot n_{s_r} + n_i \cdot n_{s_i} \cdot n_r \cdot n_{s_r} \leq n_i \cdot n_{s_i} \cdot n_r \cdot n_{s_r} \cdot 2$, when summarizing the PRHO-losses and simplifying the result. □

6.11.5. Proof

There are three separate cases:

1. The initiator-session π_i^s has no peered session.
2. The responder-session π_i^s has no peered session.
3. The session π_i^s has a peered session.

As these cases are defined to be mutually exclusive by definition and cover all possibilities, we analyze them separately and note that the adversarial advantage against our protocol is upper bounded by the sum of the adversarial advantages that can be achieved in the sub-cases.

Case A: Unpeered Initiator Session

In this first case we treat the situation where the adversary impersonates the responder (aka the satellite).

Theorem 25. *There is no polynomial-time quantum-adversary that can win the BR'-game of the triple-KEM protocol in the case where the target session is an initiator-session that has no peered session, with more than negligible probability. Specifically, we find for all such adversaries $\mathcal{A}$:*

$$\mathsf{Adv}^{BR'_{Case\ A}}_{\mathcal{A},\mathsf{3KEM}}(1^\lambda) \leq \left(\begin{array}{rrcl} & & & \mathsf{Adv}^{\textit{coll-res}}_{\mathcal{A}_1,\mathsf{H}}(1^\lambda) \\ + & n_{s_i} \cdot n_i \cdot n_r \cdot \frac{1}{1-\mathsf{RKEM}.\delta} & \cdot & \mathsf{Adv}^{\textit{IND-CCA}}_{\mathcal{A}_4,\mathsf{RKEM}}(1^\lambda) \\ + & n_{s_i} \cdot n_i \cdot n_r & \cdot & \mathsf{Adv}^{\textit{PRHO}}_{\mathsf{NHO},\mathcal{A}_5}(1^\lambda) \\ + & n_{s_i} \cdot n_i \cdot n_r & \cdot & \mathsf{Adv}^{\textit{EUF-CMA}}_{\mathcal{A}_6,\mathsf{AEAD}}(1^\lambda) \end{array} \right)$$

Proof. Let ***Game 0*** be the original BR' game.

In ***Game 1*** we abort if there is ever a hash-collision. To show that this replacement is sound, we initialize a collision-resistance-challenger for H and use whatever collision would trigger an abort to win the collision-resistance-game. Since we always win that game in case of an abort that results from the difference between *Game 0* and *Game 1* we find:

$$\Pr[\text{break}_0] \leq \Pr[\text{break}_1] + \mathsf{Adv}^{\mathsf{coll\text{-}res}}_{\mathcal{A}_1,\mathsf{H}}(1^\lambda)$$

In ***Game 2*** we guess the target-session, the party running it and the impersonated party and abort upon a wrong guess. As there are at most n_i possible initiators, n_r possible responders, and up to n_{s_i} initiator-sessions we find:

$$\Pr[\text{break}_1] \leq n_{s_i} \cdot n_i \cdot n_r \cdot \Pr[\text{break}_2]$$

In ***Game 3*** we replace the shared secret `k_sat` of the challenge-session with a random string. To show that this replacement is sound we initialize an IND-CCA-challenger for the responder-KEM and perform the following substitutions:

- Instead of generating its long-term key-pair honestly, the responder will use the challenge-public key.
- Whenever the responder needs to decapsulate a ciphertext, it uses the decapsulation-oracle of the IND-CCA-game.
- Instead of generating the ciphertext and shared secret honestly in the challenge-session, the initiator uses the challenge-ciphertext and the challenge-key.
- If any honest session recreates the challenge-ciphertext by chance, we will compare the encapsulated key to the challenge-key and use the result to win the IND-CCA-game and abort. The probability of winning in this case can be lower-bounded to $1 - \delta$, as the probability of two encapsulations of a δ-correct KEM that result in the same ciphertext for the same public key can at most be δ:

  Assume to the contrary that the probability is $\varepsilon > \delta$. Let m_0 be an honestly generated ciphertext whose encapsulation claimed that the encapsulated key was k. By the definition of a δ-correct KEM, there is now a probability of $1 - \delta$ that the decapsulation outputs the same k. By our assumption there is however also a probability ε that another invocation of the encapsulation-algorithm would result in a different shared secret k' during the encapsulation. This means that there is now a $(1 - \delta) \cdot \varepsilon$-probability of the second encapsulation decapsulating incorrectly and a probability of δ that the first decapsulated wrongly. But for $0 < \delta < \varepsilon < 1$ we have $(1-\delta) > 0$ and thus find that $(1-\delta) \cdot \varepsilon + \delta > \delta$ which is in contradiction to the assumption that the probability of a decryption-failure is at most δ.

  As a consequence, there is a slight additional loss-factor of $\frac{1}{1-\delta}$ in the reduction of this game.

- if the responder receives the challenge-ciphertext, it will use the challenge-key instead of decapsulating. Since this can only happen in a session that does not match the challenge-session (by the definition of case A) and is not partnered with an honest session (by the previous item), there is no need to modify any other session in that case.

If the challenge-bit of the IND-CCA-challenger is 0, then this is a purely conceptual change and we are in *Game 2*. Otherwise, the challenge-key is truly random and we are in *Game 3*. Thus we find:

$$\Pr[\text{break}_2] \leq \Pr[\text{break}_3] + \frac{1}{1 - \mathsf{RKEM}.\delta} \cdot \mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathcal{A}_4, \mathsf{RKEM}}(1^\lambda)$$

In ***Game 4*** we replace all outputs of the (implicit) hash-object after inputting the shared secret `k_sat` in the challenge-session and all responder-sessions that received the challenge-public-key, with random values. To show that this replacement is sound we initialize a PRHO-challenger for NHO and replace the initiator's direct use of NHO with the oracles in the following way: Whenever the initiator and the responder start a session and would normally initialize a hash-object, they will instead call Create and use the returned identifier i for all oracle invocations in that session. Whenever they would normally use the input/finalize functions of NHO it will instead invoke the In/Fin oracle, with one exception: When they would normally input `k_sat`, they will instead invoke Rand. This substitution is valid since `k_sat` is an independent random value by *Game 4*. If the challenge bit b of the PRHO game is 0, this is a purely conceptual change and we are in *Game 3*. Otherwise, all outputs after inputting `k_sat` get replaced with independent random values and we are in *Game 4*. Thus:

$$\Pr[\text{break}_3] \leq \Pr[\text{break}_4] + \mathsf{Adv}^{\mathsf{PRHO}}_{\mathsf{NHO}, \mathcal{A}_5}(1^\lambda)$$

In ***Game 5*** we abort after receiving a reply to the target-session.

To show that this replacement is sound, we initialize an EUF-CMA-challenger for AEAD and forward `c_2` to it as a potential forgery. This substitution is sound because the AEAD-key `k_1` is truly random by *Game 4* and because `c_2` is a fresh ciphertext:

There is no peered session to the challenge-session (by the definition of case A), and no collisions for the hash-function (by *Game 1*), and the expected response is the only (and therefore last) message sent by the responder, and because `c_2` cannot be a replay of `c_1`, as they use both a different nonce and a different handshake-hash, and the handshake-hash of the challenge-session is unique among all honest sessions.

This means that the associated data of `c_2` is different from the associated data used in any other honest session, which means that `c_2` has to be fresh.

If the ciphertext is not valid, we would abort in any case and thus the behavior is identical to *Game 4*. Otherwise the ciphertext is a valid forgery and we win the EUF-CMA-game and thus find:

$$\Pr\left[\text{break}_4\right] \leq \Pr\left[\text{break}_5\right] + \mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathcal{A}_6,\,\mathsf{AEAD}}\left(1^\lambda\right)$$

At this point the target-session never successfully completes, and we therefore find trivially that:

$$\Pr\left[\text{break}_5\right] = 0$$

By summarizing the losses up to this point, we find the adversarial advantage stated in Theorem 25. □

Case B: Unpeered Responder Session

In this second case we treat the situation where an attacker attempts to impersonate the initiator (aka mission-control). This is largely analogous to Case A, aside from some substitutions of instances the most notable deviation being the need to create a ciphertext for the AEAD-scheme. Furthermore, this case applies to both the Dual-KEM and the Triple-KEM protocol without any differences, as all of these differences are in parts of the protocol that have no impact here.

Theorem 26. *There is no polynomial-time quantum-adversary that can win the BR'-game of the triple-KEM protocol in the case where the target session is a responder-session that has no peered session, with more than negligible probability. Specifically, we find for all such adversaries $\mathcal{A}$:*

$$\max\left(\begin{array}{l}\mathsf{Adv}^{BR'_{Case\ A}}_{\mathcal{A},\,3\mathsf{KEM}}\left(1^\lambda\right),\\ \mathsf{Adv}^{BR'_{Case\ A}}_{\mathcal{A},\,2\mathsf{KEM}}\left(1^\lambda\right)\end{array}\right)$$

$$\leq \left(\begin{array}{rrcl} & & & \mathsf{Adv}^{coll\text{-}res}_{\mathcal{A}_1,\,\mathsf{H}}\left(1^\lambda\right)\\ + & n_{s_r}\cdot n_i\cdot n_r\cdot\frac{1}{1-\mathsf{IKEM}.\delta} & \cdot & \mathsf{Adv}^{IND\text{-}CCA}_{\mathcal{A}_4,\,\mathsf{IKEM}}\left(1^\lambda\right)\\ + & n_{s_r}\cdot n_i\cdot n_r & \cdot & \mathsf{Adv}^{PRHO}_{\mathsf{NHO},\,\mathcal{A}_5}\left(1^\lambda\right)\\ + & n_{s_r}\cdot n_i\cdot n_r & \cdot & \mathsf{Adv}^{EUF\text{-}CMA}_{\mathcal{A}_6,\,\mathsf{AEAD}}\left(1^\lambda\right)\end{array}\right)$$

Proof. Let ***Game 0*** be the original BR' game.

In ***Game 1*** we abort if there is ever a hash-collision. To show that this replacement is sound, we initialize a collision-resistance-challenger for H and use whatever collision would trigger an abort to win the collision-resistance-game. Since we always win that game in case of an abort that results from the difference between *Game 0* and *Game 1* we find:

$$\Pr[\text{break}_0] \leq \Pr[\text{break}_1] + \mathsf{Adv}^{\mathsf{coll\text{-}res}}_{\mathcal{A}_1,\mathsf{H}}(1^\lambda)$$

In ***Game 2*** we guess the target-session, the party running it and the impersonated party and abort upon a wrong guess. As there are at most n_i possible initiators, n_r possible responders, and n_{s_r} sessions we find:

$$\Pr[\text{break}_1] \leq n_{s_r} \cdot n_i \cdot n_r \cdot \Pr[\text{break}_2]$$

In ***Game 3*** we replace the shared secret `k_mc` of the challenge-session with a random string. To show that this replacement is sound we initialize an IND-CCA-challenger for the initiator-KEM and perform the following substitutions:

- Instead of generating its long-term key-pair honestly, the initiator will use the challenge-public key.
- Whenever the initiator needs to decapsulate a ciphertext, it uses the decapsulation-oracle of the IND-CCA-game.
- Instead of generating the ciphertext and shared secret honestly in the challenge-session, the responder uses the challenge-ciphertext and the challenge-key.
- If any honest session recreates the challenge-ciphertext by chance, we will compare the encapsulated key to the challenge-key and use the result to win the IND-CCA-game and abort. The probability of winning in this case can be lower-bounded to $1 - \delta$, as the probability of two encapsulations of a δ-correct KEM that result in the same ciphertext for the same public key can at most be δ:

  Assume to the contrary that the probability is $\varepsilon > \delta$. Let m_0 be an honestly generated ciphertext whose encapsulation claimed that the encapsulated key was k. By the definition of a δ-correct KEM, there is now a probability of $1 - \delta$ that the decapsulation outputs the same k. By our assumption there is however also a probability ε that another invocation of the encapsulation-algorithm would result in a different shared secret k' during the encapsulation. This means that there is now a $(1 - \delta) \cdot \varepsilon$-probability of the second encapsulation decapsulating incorrectly and a probability of δ that the first decapsulated wrongly. But for $0 < \delta < \varepsilon < 1$ we find that $(1 - \delta) \cdot \varepsilon + \delta > \delta$ which is in contradiction to the assumption that the probability of a decryption-failure is at most δ.

As a consequence, there is a slight additional loss-factor of $\frac{1}{1-\delta}$ in the reduction of this game.

- If the initiator receives the challenge-ciphertext, it will use the challenge-key instead of decapsulating. Since this can only happen in a session that does not match the challenge-session (by the definition of case B) and is not partnered with an honest session (by the previous item), there is no need to modify any other session in that case.

If the challenge-bit of the IND-CCA-challenger is 0, then this is a purely conceptual change and we are in *Game 2*. Otherwise, the challenge-key is truly random, and we are in *Game 3*. Thus, we find:

$$\Pr[\text{break}_2] \leq \Pr[\text{break}_3] + \frac{1}{1-\mathsf{IKEM}.\delta} \cdot \mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathcal{A}_4,\mathsf{IKEM}}(1^\lambda)$$

In ***Game 4*** we replace all outputs of the (implicit) hash-object after inputting the shared secret `k_mc` in the challenge-session and all responder-sessions that received the challenge-public-key, with random values. To show that this replacement is sound we initialize a PRHO-challenger for NHO and replace the initiator's direct use of NHO with the oracles in the following way: Whenever the initiator and the responder start a session and would normally initialize a hash-object, they will instead call Create and use the returned identifier i for all oracle invocations in that session. Whenever they would normally use the input/finalize functions of NHO it will instead invoke the In/Fin oracle, with one exception: When they would normally input `k_mc`, they will instead invoke Rand. This substitution is valid since `k_mc` is an independent random value by *Game 4*. If the challenge bit b of the PRHO game is 0, this is a purely conceptual change and we are in *Game 3*. Otherwise, all outputs after inputting `k_mc` get replaced with independent random values and we are in *Game 4*. Thus:

$$\Pr[\text{break}_3] \leq \Pr[\text{break}_4] + \mathsf{Adv}^{\mathsf{PRHO}}_{\mathsf{NHO},\mathcal{A}_5}(1^\lambda)$$

In ***Game 5*** we abort after receiving a reply to the target-session's message.

To show that this replacement is sound, we initialize an EUF-CMA-challenger for AEAD perform the following substitutions:

- The responder will disregard `k_3` and instead use the encryption-oracle provided by the EUF-CMA-game to compute `c_4` if necessary (`c_4` encrypts the updated long-term key `pk_sat_new` and is thus never needed in Dual-KEM).

- Upon receiving `c_5`, we forward it to the EUF-CMA-challenger as a potential forgery.

This substitution is sound because the AEAD-key `k_3` is truly random by *Game 4* and because `c_5` cannot be a replay of `c_4`, as they use both a different nonce and a different handshake-hash as associated data by *Game 1*.

As there is no peered session to the challenge-session (by the definition of case B), and no collisions for the hash-function (by *Game 1*), and the expected response is the last message in the handshake, the handshake-hash of the challenge-session is unique among all honest sessions.

If the ciphertext is not valid, we would abort in any case and thus the behavior is identical to *Game 4*. Otherwise, the ciphertext is a valid forgery and we win the EUF-CMA-game and thus find:

$$\Pr\left[\text{break}_4\right] \leq \Pr\left[\text{break}_5\right] + \mathsf{Adv}^{\mathsf{AEAD}}_{\mathcal{A}_5,\mathsf{AEAD}}\left(1^\lambda\right)$$

At this point the target-session never successfully completes, and we therefore find trivially that:

$$\Pr\left[\text{break}_5\right] = 0$$

By summarizing the losses up to this point, we find the adversarial advantage stated in Theorem 26. □

Case C: Peered Session

This last case intuitively treats adversaries that attempt to break the confidentiality of the protocol.

Theorem 27. *There is no polynomial-time quantum-adversary that can win the BR'-game of the triple-KEM protocol in the case where the target session has a peered session, with more than negligible probability. Specifically we find for all such adversaries $\mathcal{A}$:*

$$\max\left(\begin{array}{l}\mathsf{Adv}^{\mathit{BR'}_{\mathit{Case\ C}}}_{\mathcal{A},\mathsf{3KEM}}\left(1^\lambda\right),\\ \mathsf{Adv}^{\mathit{BR'}_{\mathit{Case\ C}}}_{\mathcal{A},\mathsf{2KEM}}\left(1^\lambda\right)\end{array}\right)$$

$$\leq\left(\begin{array}{lrcl} & & & \mathsf{Adv}^{\mathit{coll\text{-}res}}_{\mathcal{A}_1,\mathsf{H}}\left(1^\lambda\right)\\ + & n_i\cdot n_{s_i} & \cdot & \mathsf{EKEM}.\delta\\ + & n_i\cdot n_{s_i}\cdot n_r\cdot n_{s_r}\cdot 3 & \cdot & \mathsf{Adv}^{\mathit{IND\text{-}CCA}}_{\mathcal{A},\mathsf{EKEM}}\left(1^\lambda\right)\\ + & n_i\cdot n_{s_i}\cdot n_r\cdot n_{s_r} & \cdot & \mathsf{Adv}^{\mathit{PRHO}}_{\mathcal{A},\mathsf{NHO}}\left(1^\lambda\right)\\ + & n_i\cdot n_{s_i}\cdot n_r\cdot n_{s_r}\cdot 2 & \cdot & \mathsf{Adv}^{\mathit{PRF}}_{\mathcal{A},\mathsf{KDF}}\left(1^\lambda\right)\end{array}\right)$$

Proof. Let ***Game 0*** be the original BR' game.

In ***Game 1*** we abort if there is ever a hash-collision. To show that this replacement is sound, we initialize a collision-resistance-challenger for H and use whatever collision would trigger an abort to win the collision-resistance-game. Since we always win that game in case of an abort that results from the difference between *Game 0* and *Game 1* we find:

$$\Pr\left[\mathrm{break}_0\right]\leq\Pr\left[\mathrm{break}_1\right]+\mathsf{Adv}^{\mathsf{coll\text{-}res}}_{\mathcal{A}_1,\mathsf{H}}\left(1^\lambda\right)$$

In ***Game 2*** we guess the initiator of the target-interaction as well as the initiator-session of that interaction and abort upon a wrong guess. Because there are n_i possible initiators who initiate up to n_{s_i} sessions each we find:

$$\Pr\left[\mathrm{break}_1\right]\leq n_i\cdot n_{s_i}\cdot\Pr\left[\mathrm{break}_2\right]$$

In ***Game 3*** we abort if there is ever an honest session that shares the public key with the initiator-session. To show that this substitution is valid we initialize an $\mathsf{IND\text{-}CCA1}$-challenger for EKEM and embed the challenge public-key into the initiators target-session, using the decapsulation-oracle for decapsulation. If we now honestly recreate that public key in a different session, we can use the secret key to break the security of EKEM: Because we assume that the probability of an decapsulation-failure with honestly generated ciphertexts (such as the challenge-ciphertext) of EKEM is at most $\mathsf{EKEM}.\delta$ and because the ciphertexts depends only on the public key and independent randomness, the secret key of the colliding public key is a (or sometimes: the) matching secret key for the challenge public key. (We remark that this doesn't even just break IND, but is in fact a full key-extraction attack.) We thus find:

$$\Pr\left[\mathrm{break}_2\right]\leq\mathsf{Adv}^{\mathsf{IND\text{-}CCA1}}_{\mathcal{B},\mathsf{EKEM}}\left(1^\lambda\right)+\mathsf{EKEM}.\delta+\Pr\left[\mathrm{break}_3\right]$$

In ***Game 4*** we guess the responder of the target-interaction and the targeted responder-session and abort upon a wrong guess. As there are n_r potential responders and n_{s_r} responder-sessions we find:

$$\Pr[\text{break}_3] \leq n_r \cdot n_{s_r} \cdot \Pr[\text{break}_4]$$

In ***Game 5*** we abort if there is ever an honest responder-session that is not part of the target-interaction that produces the same ciphertext for the EKEM that is used in the challenge-interaction for the same public-key that is used in the challenge-interaction. (By *Game 3* there is only one honest initiator-session that uses that public key, but there may be dishonest ones in addition.) To show that this substitution is sound we initialize an IND-CPA-challenger for EKEM and again replace the public key used in the initiator's challenge-session with the challenge public-key. Furthermore, we replace the ciphertext used in the challenge-session with the challenge-ciphertext and the resulting key with the challenge-key. If any later honest session recreates the challenge-ciphertext, then the contained key can be used to trivially win the IND-CPA-game. (We remark that this is in fact a full plaintext-recovery attack.) We thus find:

$$\Pr[\text{break}_4] \leq \mathsf{Adv}^{\mathsf{IND\text{-}CPA}}_{\mathcal{B},\,\mathsf{EKEM}}(1^\lambda) + \Pr[\text{break}_5]$$

In ***Game 6*** we abort if there is ever an honest session that is not part of the target-interaction but shares the handshake-hash `h_6` with it.

This is perfectly indistinguishable from *Game 5*, because *Game 1* ensures that there are no hash-collisions, *Game 3* ensures that the EKEM public key of the target-interaction is different from that in all other honest initiator-sessions, and *Game 5* ensures that the EKEM-ciphertext in the challenge interaction is different from that in all other honest responder-sessions. Since both of these values make their way into the handshake-hash and since any interaction that involves at least one honest party will therefore add at least one value that is different from the one in the challenge-interaction we can conclude that the handshake-hash of the challenge-interaction is not repeated in any honest sessions.

Because this is a purely conceptual change at this point, we find:

$$\Pr[\text{break}_5] = \Pr[\text{break}_6]$$

In ***Game 7*** we replace the shared secret `k_e` that is encapsulated for the target-session's ephemeral key and fed into the pseudo-random hash-object with a random string. To show that this replacement is sound we initialize

an IND-CCA-challenger for the ephemeral KEM and perform the following substitutions:

Instead of generating the ephemeral keypair for the target-session himself, the initiator will use the challenge-public key of the IND-CCA-game.

Instead of generating the KEM-ciphertext honestly, the responder will use the challenge-ciphertext of the IND-CCA-game.

Both parties will use the challenge-key of the IND-CCA-game in place of any encapsulated keys.

In the event that the initiator receives a different ciphertext than the challenge-ciphertext (due to adversarial actions), he will use the decapsulation-oracle of the IND-CCA-game.

If the challenge-bit of the IND-CCA-game is 0, this is a purely conceptual change, and we are in *Game 6*. Otherwise, the challenge-key is a truly random string, and we are in *Game 7* and thus find:

$$\Pr\left[\text{break}_6\right] \leq \Pr\left[\text{break}_7\right] + \mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathcal{A},\,\mathsf{EKEM}}\left(1^\lambda\right)$$

In ***Game 8*** we replace the outputs of the pseudo-random hash-object, after feeding it the EKEM-key, with random strings. To show that this replacement is sound we initialize a PRHO-challenger for NHO and perform the following substitutions:

Whenever the initiator and the responder start a session and would normally initialize a hash-object, they will instead call Create and use the returned identifier i for all oracle invocations in that session. Whenever they would normally use the $\mathsf{input}/\mathsf{finalize}$ functions of NHO it will instead invoke the In/Fin oracle, with one exception: When they would normally input `k_e`, they will instead invoke Rand. This substitution is valid since `k_e` is an independent random value by *Game 7*. If the challenge bit b of the PRHO game is 0, this is a purely conceptual change and we are in *Game 7*. Otherwise, all outputs after inputting `k_e` get replaced with independent random values and we are in *Game 8*. Thus:

$$\Pr\left[\text{break}_7\right] \leq \Pr\left[\text{break}_8\right] + \mathsf{Adv}^{\mathsf{PRHO}}_{\mathcal{A},\,\mathsf{NHO}}\left(1^\lambda\right)$$

In ***Game 9*** we replace the final keys with random strings. To show that this replacement is sound we will use two sub-games.

In ***Game 9.A*** we replace `key_mc` with a random string. To show that this replacement is sound we initialize a PRF-challenger for KDF and replace the evaluation of KDF using `pre_key_mc` as key with an invocation of the challenge-oracle. This substitution is sound because `pre_key_mc` is truly random by *Game 8* and because `h_6` is a different message than used in any

other session by *Game 6*, including sessions where the peer is not honest. If the challenge-bit is 0, then this is therefore a purely conceptual change, and we are in *Game 8*. Otherwise `key_mc` is a truly random string and we are in *Game 9.A* and find:

$$\Pr\left[\text{break}_8\right] \leq \Pr\left[\text{break}_{9.A}\right] + \mathsf{Adv}^{\mathsf{PRF}}_{\mathcal{A},\mathsf{KDF}}\left(1^\lambda\right)$$

In ***Game 9.B*** we replace `key_sat` with a random string. To show that this replacement is sound we initialize a PRF-challenger for KDF and replace the evaluation of KDF using `pre_s_g` as key with an invocation of the challenge-oracle. This substitution is sound because `pre_s_g` is truly random by *Game 8* and because `h_6` is a different message than used in any other session by *Game 6*, including sessions where the peer is not honest. If the challenge-bit is 0, then this is therefore a purely conceptual change, and we are in *Game 9.A*. Otherwise `key_mc` is a truly random string and we are in *Game 9.B* and find:

$$\Pr\left[\text{break}_{9.A}\right] \leq \Pr\left[\text{break}_{9.B}\right] + \mathsf{Adv}^{\mathsf{PRF}}_{\mathcal{A},\mathsf{KDF}}\left(1^\lambda\right)$$

By noting that *Game 9.B*=*Game 9* and summarizing the losses in the sub-games we find:

$$\Pr\left[\text{break}_8\right] \leq \Pr\left[\text{break}_9\right] + 2 \cdot \mathsf{Adv}^{\mathsf{PRF}}_{\mathcal{A},\mathsf{KDF}}\left(1^\lambda\right)$$

At this point we note that the final keys are originally outputs of the PRHO-challenger and thus random; therefore, the adversary has to distinguish a random string from a random string which trivially limits the adversarial success-probability to $\frac{1}{2}$ with an advantage of 0:

$$\Pr\left[\text{break}_9\right] = 0$$

By summarizing the security-losses in the previous games and noting that $n_i \cdot n_{s_i} \cdot \left(\mathsf{Adv}^{\mathsf{IND\text{-}CCA1}}_{\mathcal{B},\mathsf{EKEM}}\left(1^\lambda\right) + n_r \cdot n_{s_r} \cdot \left(\mathsf{Adv}^{\mathsf{IND\text{-}CPA}}_{\mathcal{B},\mathsf{EKEM}}\left(1^\lambda\right) + \mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathcal{A},\mathsf{EKEM}}\left(1^\lambda\right)\right)\right) \leq n_i \cdot n_{s_i} \cdot n_r \cdot n_{s_r} \cdot 3 \cdot \mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathcal{A},\mathsf{EKEM}}\left(1^\lambda\right)$ to simplify the resulting equation, we find the adversarial advantage claimed in Theorem 27. □

7. Conclusion

In the first major part of this thesis on cryptographic protocols we have shown that strong forms of plausible deniability are achievable with signature-based authentication. This resolved the threat of having a social question, whether deniability is a desirable property, being decided based on technical possibility instead of social desirability. This comes with the caveat however, that this opens that second question of whether it is socially desirable, which is a question that we do not feel qualified to answer scientifically. We therefore merely observe that at this time the trend in opinion seems to be moving towards viewing it as undesirable, but note that this should not be taken as a scientific statement, nor as a way to answer that question comprehensively.

We have furthermore shown as the primary result that transforming pre-quantum protocols, especially those based on the Noise-Framework, to protocols that provide post-quantum security is often practical and does not have to come at extremely high costs to real world performance.

That being said, some challenges remain: No post-quantum primitive that is currently considered for standardization is able to match all size- and performance characteristics of elliptic curve based cryptography.

Furthermore most of the current focus has been on key-exchange and messaging protocols, but there also exists a significant body of other protocols that used algebraic properties of certain pre-quantum primitives. Replacing these will necessarily be harder and likely require a much closer integration of the primitives themselves, though some of the algebraic properties of especially lattice-based cryptography are likely able to provide many of the required properties. Turning them into protocols that are usable and secure in practice is however still going to be a significant challenge for years to come.

A. Post Quantum WireGuard

A.1. Security-Model

The following section is only included for reference and essentially copied verbatim from the Wireguard-analysis by Dowling and Paterson [DP18] with the only change being that this version explicitly considers quantum-adversaries. All our changes are highlighted in the same way as this paragraph.

We propose a modification to the eCK-PFS security model introduced by Cremers and Feltz [CF12] that incorporates pre-shared keys and strengthens the security definitions accordingly. We explain the framework and give an algorithmic description of the security model in Section A.1, and describe the corruption abilities of the adversary in Section A.1. We then describe the modifications necessary to capture the exact security guarantees that WireGuard attempts to achieve by explaining the differences between our partnering definitions and traditional notions of partnering in Section A.1. We then give our modified cleanness definitions in Section A.1. Given that WireGuard uses a mix of long-term identity keys, ephemeral keys and pre-shared secrets in its key exchange protocol, it is appropriate to use an extended-Canetti-Krawcyzk model (as introduced in [LLM07]), wherein the adversary is allowed to reveal subsets of these secrets. It is claimed in [Don17] that WireGuard "achieves the requirements of authenticated key exchange (AKE) security, avoids key-compromise impersonation, avoids replay attacks, provides perfect forward secrecy," [Don17]. These are all notions captured by our extended eCK-PFS model, so our subsequent security proof will formally establish that WireGuard meets its goals.

Execution Environment

Consider an experiment $\mathsf{Exp}^{\mathsf{eCK\text{-}PFS\text{-}PSK}}_{\mathsf{KE},n_P,n_S,\mathcal{A}}(\lambda)$ played between a challenger $\mathcal{C}$ and an adversary $\mathcal{A}$. $\mathcal{C}$ maintains a set of n_P parties $P_1, \ldots, P_{n_P}$ (representing users interacting with each other via the protocol), each capable of running up to n_S sessions of a probabilistic key-exchange protocol KE, represented as a tuple of algorithms $\mathsf{KE} = (f, \mathsf{ASKeyGen}, \mathsf{PSKeyGen}, \mathsf{EPKeyGen})$. We use

π_i^s to refer to both the identifier of the s-th instance of the KE being run by party P_i and the collection of per-session variables maintained for the s-th instance of KE run by P_i. We describe the algorithms below:

$\mathsf{KE}.f(\lambda, pk_i, sk_i, \pi, m) \rightarrow_\$ (m', \pi')$ is a (potentially) probabilistic algorithm that takes a security parameter λ, the long-term asymmetric key pair pk_i, sk_i of the party P_i, a collection of per-session variables π and an arbitrary bit string $m \in \{0,1\}^* \cup \{\emptyset\}$, and outputs a response $m' \in \{0,1\}^* \cup \{\emptyset\}$ and an updated per-session state π', acting in accordance with an honest protocol implementation.

$\mathsf{KE.ASKeyGen}(\lambda) \rightarrow_\$ (pk, sk)$ is a probabilistic asymmetric-key generation algorithm taking as input a security parameter λ and outputting a public-key/secret-key pair (pk, sk).

$\mathsf{KE.PSKeyGen}(\lambda) \rightarrow_\$ (psk, pskid)$ is a probabilistic symmetric-key generation algorithm that also takes as input a security parameter λ and outputs a symmetric pre-shared secret key psk and (potentially) a pre-shared secret key identifier $pskid$.

$\mathsf{KE.EPKeyGen}(\lambda) \rightarrow_\$ (ek, epk)$ is a probabilistic ephemeral-key generation algorithm that also takes as input a security parameter λ and outputs an asymmetric public-key/secret-key pair (ek, epk).

$\mathcal{C}$ runs $\mathsf{KE.ASKeyGen}(\lambda)$ n_P times to generate a public-key/secret-key pair (pk_i, sk_i) for each party $P_i \in \{P_1, \ldots, P_{n_P}\}$ and delivers all public-keys pk_i for $i \in \{1, \ldots, n_P\}$ to $\mathcal{A}$. The challenger $\mathcal{C}$ then randomly samples a bit $b \xleftarrow{\$} \{0,1\}$ and interacts with the adversary via the queries listed in Section A.1. Eventually, $\mathcal{A}$ terminates and outputs a guess b' of the challenger bit b. The adversary wins the $\mathsf{eCK\text{-}PFS\text{-}PSK}$ key-indistinguishability experiment if $b' = b$, and additionally if the session π_i^s such that $\mathsf{Test}(i, s)$ was issued satisfies a cleanness predicate clean, which we discuss in more detail in Section A.1. We give an algorithmic description of this experiment in Figure A.1.

Each session maintains the following set of per-session variables:

- $\rho \in \{\texttt{init}, \texttt{resp}\}$ – the role of the party in the current session. Note that parties can be directed to act as `init` or `resp` in concurrent or subsequent sessions.
- $pid \in \{1, \ldots, n_P, \star\}$ – the intended communication partner, represented with $\star$ if unspecified. Note that the identity of the partner session may be set during the protocol execution, in which case pid can be updated once.
- $m_s \in \{0,1\}^* \cup \{\perp\}$ – the concatenation of messages sent by the session, initialized by $\perp$.

$\mathsf{Exp}^{\mathsf{eCK\text{-}PFS\text{-}PSK\text{-}ind}}_{\mathsf{KE},\mathsf{clean},n_P,n_S,\mathcal{A}}(()\,\lambda)$:

$b \xleftarrow{\$} \{0,1\}$
tested $\leftarrow$ `false`
for $i = 1$ **to** n_P **do**
  $(pk_i, sk_i) \xleftarrow{\$} \mathsf{ASKeyGen}(\lambda)$
  $\mathsf{ASKflag}_i \leftarrow$ `clean`
  $\vec{\mathsf{PSK}}_i[1], \ldots, \vec{\mathsf{PSK}}_i[n_P] \leftarrow \perp$
  $\vec{\mathsf{PSKflag}}_i[1], \ldots, \vec{\mathsf{PSKflag}}_i[n_P] \leftarrow \perp$
  $\vec{\mathsf{EPKflag}}_i[1], \ldots, \vec{\mathsf{EPKflag}}_i[n_S] \leftarrow \perp$
  $\vec{\mathsf{RSKflag}}_i[1], \ldots, \vec{\mathsf{RSKflag}}_i[n_S] \leftarrow \perp$
  $ct_i \leftarrow 0$
end for
$b' \xleftarrow{\$} \mathcal{A}^{\mathsf{Send},\mathsf{Create}*,\mathsf{Corrupt}*,\mathsf{Reveal},\mathsf{Test}}(pk_1, \ldots, pk_{n_P})$
if $\mathsf{clean}(\pi_i^s) \wedge (b = b')$ **then**
  return 1
else
  $b' \xleftarrow{\$} \{0,1\}$
  return b'
end if

$\mathsf{Create}(i, j, role)$:

$ct_i \leftarrow ct_i + 1$
$s \leftarrow ct_i$
$\pi_i^s.pid \leftarrow j$
$\pi_i^s.\rho \leftarrow role$
$\pi_i^s.ek \leftarrow \mathsf{KE.EPKeyGen}(\lambda)$
$\pi_i^s.psk \leftarrow \vec{\mathsf{PSK}}_i[j]$
return (i, s)

$\mathsf{Send}(i, s, m)$:

if $\pi_i^s = \perp$ **then**
  return $\perp$
else
  $\pi_i^s.m_r \leftarrow \pi_i^s.m_r \| m$
  $(\pi_i^s, m') \leftarrow \mathsf{KE}.f(\lambda, pk_i, sk_i, \pi_i^s, m)$
  $\pi_i^s.m_s \leftarrow \pi_i^s.m_s \| m'$
  $\pi_i^s.T \leftarrow \pi_i^s.T \| m \| m'$
  return m'
end if

$\mathsf{CorruptASK}(i)$:

$\mathsf{ASKflag}_i \leftarrow$ `corrupt`
return sk_i

$\mathsf{CreatePSK}(i, j)$:

if $(i = j) \vee (\vec{\mathsf{PSKflag}}_i[j] \neq \perp)$ **then**
  return $\perp$
end if
$(psk, pskid) \leftarrow \mathsf{KE.PSKeyGen}(\lambda)$
$\vec{\mathsf{PSK}}_i[j] \leftarrow (psk, pskid)$
$\vec{\mathsf{PSK}}_j[i] \leftarrow (psk, pskid)$
$\vec{\mathsf{PSKflag}}_i[j], \vec{\mathsf{PSKflag}}_j[i] \leftarrow$ `clean`
if $pskid \neq \emptyset$ **then**
  return $pskid$
else
  return $\top$
end if

$\mathsf{Reveal}(i, s)$:

if $\pi_i^s.\alpha \neq$ `accept` **then**
  return $\perp$
else
  $\vec{\mathsf{RSKflag}}_i[s] \leftarrow$ `corrupt`
  return $\pi_i^s.k$
end if

$\mathsf{CorruptEPK}(i, s)$:

$\vec{\mathsf{EKflag}}_i[s] \leftarrow$ `corrupt`
return $\pi_i^s.ek$

$\mathsf{CorruptPSK}(i, j)$:

if $\vec{\mathsf{PSK}}_i[j] = \perp$ **then**
  return $\perp$
end if
if $\vec{\mathsf{PSKflag}}_i[j] \neq$ `clean` **then**
  return $\perp$
else
  $\vec{\mathsf{PSKflag}}_i[j] \leftarrow$ `corrupt`
  $\vec{\mathsf{PSKflag}}_j[i] \leftarrow$ `corrupt`
  return $\vec{\mathsf{PSK}}_i[j]$
end if

$\mathsf{Test}(i, s)$:

if (tested $=$ `true`) $\vee$ ($\pi_i^s.\alpha \neq$ `accept`) **then**
  return $\perp$
end if
tested $\leftarrow$ `true`
if $b = 0$ **then**
  return $\pi_i^s.k$
else
  $k \xleftarrow{\$} \mathcal{K}$
  return k
end if

Figure A.1.: eCK-PFS-PSK experiment for adversary $\mathcal{A}$ against the key-indistinguishability security of protocol KE.

- $m_r \in \{0,1\}^* \cup \{\perp\}$ – the concatenation of messages received by the session, initialized by $\perp$.
- $kid \in \{0,1\}^* \cup \{\perp\}$ – the concatenation of public keyshare information received by the session, initialized by $\perp$.
- $\alpha \in \{\texttt{active}, \texttt{accept}, \texttt{reject}, \perp\}$ – the current status of the session, initialized with $\perp$.
- $k \in \{0,1\}^* \cup \{\perp\}$ – the computed session key, or $\perp$ if no session key has yet been computed.
- $ek \in \{0,1\}^* \times \{0,1\}^* \cup \{\perp\}$ – the ephemeral key pair used by the session during protocol execution, initialized as $\perp$.
- $psk \in \{0,1\}^* \times \{0,1\}^* \cup \{\perp\}$ – the pre-shared secret and identifier used by the session during protocol execution, initialized as $\perp$.
- $st \in \{0,1\}^*$ – any additional state used by the session during protocol execution.

Finally, the challenger manages the following set of corruption registers, which hold the leakage of secrets that $\mathcal{A}$ has revealed.

- pre-shared keys $\{\mathsf{PS\vec{K}flag}_i, \mathsf{PS\vec{K}flag}_2, \ldots, \mathsf{PS\vec{K}flag}_{n_P}\}$
  where for each element
  $\mathsf{PS\vec{K}flag}_i[j] \in \mathsf{PS\vec{K}flag}_i$, $\mathsf{PS\vec{K}flag}_i[j] \in \{\texttt{corrupt}, \texttt{clean}, \perp\}\ \forall\ i, j \in [n_P]$
  and $\mathsf{PS\vec{K}flag}_i[j] = \perp$ for $i = j$
- long-term keys $\{\mathsf{SKflag}_1, \ldots, \mathsf{SKflag}_{n_P}\}$, where
  $\mathsf{SKflag}_i \in \{\texttt{corrupt}, \texttt{clean}, \perp\}\ \forall\ i \in [n_P]$
- ephemeral keys $\{\mathsf{EK\vec{f}lag}_1, \ldots, \mathsf{EK\vec{f}lag}_{n_P}\}$, where
  $\mathsf{EK\vec{f}lag}_i[s] \in \{\texttt{corrupt}, \texttt{clean}, \perp\}\ \forall\ i \in [n_P]$ and $s \in [n_S]$.

We formalize the advantage of a (potentially quantum) algorithm $\mathcal{A}$ in winning the eCK-PFS-PSK key indistinguishability experiment in the following way:

Definition 51 (eCK-PFS-PSK Key Indistinguishability)**.** Let KE be a key-exchange protocol, and n_P, $n_S \in \mathbb{N}$. For a particular given predicate clean,

and a (potentially quantum) algorithm $\mathcal{A}$, we define the advantage of $\mathcal{A}$ in the eCK-PFS-PSK key-indistinguishability game to be:

$$\mathsf{Adv}^{\mathsf{eCK\text{-}PFS\text{-}PSK,clean}}_{\mathsf{KE},n_P,n_S,\mathcal{A}}(\lambda) = |\Pr[\mathsf{Exp}^{\mathsf{eCK\text{-}PFS\text{-}PSK,clean}}_{\mathsf{KE},n_P,n_S,\mathcal{A}}(()\,\lambda) = 1] - \frac{1}{2}|.$$

We say that KE is eCK-PFS-PSK-secure if, for all $\mathcal{A}$ in QPT, $\mathsf{Adv}^{\mathsf{eCK\text{-}PFS\text{-}PSK,clean}}_{\mathsf{KE},n_P,n_S,\mathcal{A}}(\lambda)$ is negligible in the security parameter λ.

Adversarial Interaction

Our security model is intended to be as generic as possible, in order to capture eCK-like security notions, but to also include long-term pre-shared keys. This would allow our model to be used in analyzing (for example) the Signal protocol, where users exchange both long-term Diffie-Hellman keyshares used in many protocol executions, but also many ephemeral Diffie-Hellman keyshares that are only used within a single session. Another example would be TLS 1.3, where users may have established pre-shared keys to reduce the protocol's computational overheads, or to enable 0-RTT confidential data transmission.

Our attacker is a standard key-exchange model adversary, in complete control of the communication network, able to modify, inject, delete or delay messages. They can also compromise several layers of secrets:

- long-term private keys, modeling the misuse or corruption of long-term secrets in other sessions, and additionally allowing our model to capture forward-secrecy notions.
- ephemeral private keys, modeling the use of bad randomness generators.
- pre-shared symmetric keys, modeling the leakage of shared secrets, potentially due to the misuse of the pre-shared secret by the partner, or the forced later revelation of these keys.
- session keys, modeling the leakage of keys by their use in bad cryptographic algorithms.

The adversary interacts with the challenger via the queries below. An algorithmic description of how the challenger responds is in Figure A.1.

- $\mathsf{Create}(i, j, role) \to \{(i, s), \perp\}$: allows the adversary to begin new sessions. The challenger $\mathcal{C}$ creates a new session π^s_i with $\pi^s_i.pid \leftarrow j$, $\pi^s_i.\rho \leftarrow role$, $\pi^s_i.\alpha \leftarrow \texttt{active}$, $\pi^s_i.T \leftarrow \perp$, $\pi^s_i.sid \leftarrow \perp$, $\pi^s_i.k \leftarrow \perp$. $\mathcal{C}$

also computes $\mathsf{KE.EKeyGen}(\lambda) \rightarrow_{\$} (ek, epk)$ and sets $\pi_i^s.ek \leftarrow ek$. If a session π_i^s has already been created, $\mathcal{C}$ returns $\perp$. Otherwise, $\mathcal{C}$ returns (i, s) to $\mathcal{A}$.

- $\mathsf{CreatePSK}(i, j) \rightarrow \{pskid, \top, \perp\}$: allows the adversary to direct parties to generate a pre-shared key for use in future protocol executions. The challenger $\mathcal{C}$ checks that $i \neq j$ and that $\vec{\mathsf{PSK}}_i[j] = \vec{\mathsf{PSK}}_j[i] = \perp$. $\mathcal{C}$ then computes $\mathsf{KE.PSKeyGen}(\lambda) \rightarrow_{\$} psk$ and sets $\vec{\mathsf{PSK}}_i[j] = \vec{\mathsf{PSK}}_j[i] \leftarrow psk$, and the PSK register $\vec{\mathsf{PSKflag}}_i[j] = \vec{\mathsf{PSKflag}}_j[i] \leftarrow \texttt{clean}$. If $pskid \neq \emptyset$, then $\mathcal{C}$ returns $pskid$ to $\mathcal{A}$, otherwise $\mathcal{C}$ returns $\top$ (where $\top$ is a generic success flag) to $\mathcal{A}$. If $\vec{\mathsf{PSK}}_i[j] \neq \perp$ or $\vec{\mathsf{PSK}}_j[i] \neq \perp$ (i.e. if $\mathcal{A}$ has previously issued a $\mathsf{CreatePSK}(i, j)$ or $\mathsf{CreatePSK}(j, i)$ query), then $\mathcal{C}$ returns $\perp$ to $\mathcal{A}$.

- $\mathsf{Reveal}(i, s)$: allows the adversary access to the secret session key computed by a session during protocol execution. The challenger checks whether the cleanness of the session π_i^s has been upheld and $\pi_i^s.\alpha = \texttt{accept}$ and if so, returns $\pi_i^s.k$ to $\mathcal{A}$. Otherwise, $\mathcal{C}$ returns $\perp$ to $\mathcal{A}$.

- $\mathsf{CorruptPSK}(i) \rightarrow \{psk, \perp\}$: allows the adversary access to the secret pre-shared key jointly shared by parties prior to protocol execution. The challenger $\mathcal{C}$ checks that $\vec{\mathsf{PSK}}_i[j] = \vec{\mathsf{PSK}}_j[i] \neq \perp$, and that $\vec{\mathsf{PSKflag}}_i[j] = \vec{\mathsf{PSKflag}}_j[i] = \texttt{clean}$. If so, $\mathcal{C}$ returns $PSK \leftarrow \vec{\mathsf{PSK}}_i[j]$ to $\mathcal{A}$ and sets $\vec{\mathsf{PSKflag}}_i[j] = \vec{\mathsf{PSKflag}}_j[i] \leftarrow \texttt{corrupt}$. If $\vec{\mathsf{PSK}}_i[j] = \vec{\mathsf{PSK}}_j[i] = \perp$ or $\vec{\mathsf{PSKflag}}_i[j] = \vec{\mathsf{PSKflag}}_j[i] \neq \texttt{clean}$, (i.e. that the adversary has either not previously created a psk between the two parties P_i and P_j, or has previously issued a $\mathsf{CorruptPSK}(i, j)$/$\mathsf{CorruptPSK}(j, i)$ query), then $\mathcal{C}$ returns $\perp$ to $\mathcal{A}$.

- $\mathsf{CorruptASK}(i) \rightarrow \{sk_i, \perp\}$: allows the adversary access to the secret long-term key generated by a party prior to protocol execution. The challenger $\mathcal{C}$ checks that $\mathsf{ASKflag}_i \neq \texttt{corrupt}$. If so, $\mathcal{C}$ returns sk_i to $\mathcal{A}$. If $\mathsf{ASKflag}_i = \texttt{corrupt}$ (i.e. $\mathcal{A}$ has previously issued a $\mathsf{CorruptASK}(i)$ query), then $\mathcal{C}$ returns $\perp$ to $\mathcal{A}$.

- $\mathsf{CorruptEPK}(i, s) \rightarrow \{ek, \perp\}$: allows the adversary access to the secret ephemeral key generated by a session during protocol execution. The challenger $\mathcal{C}$ checks that $\vec{\mathsf{EPKflag}}_{i,s} = \texttt{clean}$. If so, $\mathcal{C}$ returns $\pi_i^s.ek$ to $\mathcal{A}$, and sets $\vec{\mathsf{EPKflag}}_{i,s} \leftarrow \texttt{corrupt}$. If $\vec{\mathsf{EPKflag}}_{i,s} = \texttt{corrupt}$, (i.e. $\mathcal{A}$ has previously issued a $\mathsf{CorruptEPK}(i, s)$ query), then $\mathcal{C}$ returns $\perp$ to $\mathcal{A}$.

- $\mathsf{Send}(i, s, m) \to \{m', \perp\}$: allows the adversary to send messages to sessions for protocol execution and receive their output. If a session π_i^s has not been previously created, or $\pi_i^s.\alpha \neq \texttt{active}$, then $\mathcal{C}$ returns $\perp$ to $\mathcal{A}$. Otherwise, $\mathcal{C}$ computes $\mathsf{KE}.f(\lambda, m, \pi_i^s) \to (m', \pi_i^s)$, sets $\pi_i^s \leftarrow \pi_i^{s'}$, and returns m' to $\mathcal{A}$.
- $\mathsf{Test}(i, s) \to \{k, \perp\}$: sends the adversary a real-or-random session key used in determining the success of $\mathcal{A}$ in the key-indistinguishability game. If a session π_i^s exists and $\pi_i^s.\alpha = \texttt{accept}$, then the challenger $\mathcal{C}$ samples a key $k_0 \xleftarrow{\$} \mathcal{D}$ where $\mathcal{D}$ is the distribution of the session key, and sets $k_1 \leftarrow \pi_i^s.k$. $\mathcal{C}$ then returns k_b (where b is the random bit sampled during set-up) to $\mathcal{A}$. If a session π_i^s does not exist, or $\pi_i^s.\alpha \neq \texttt{accept}$, then $\mathcal{C}$ returns $\perp$ to $\mathcal{A}$.

Partnering Definitions

In order to evaluate which secrets the adversary is able to reveal without trivially breaking the security of the protocol, key-exchange models must define how sessions are *partnered*. Otherwise, an adversary would simply run a protocol between two sessions, faithfully delivering all messages, Test the first session to receive the real-or-random key, and Reveal the session partner's key. If the keys are equal, then the Test key is real, and otherwise the session key has been sampled randomly. BR-style key-exchange models traditionally use *matching conversations* in order to do this. When introducing the eCK-PFS model, Cremers and Feltz [CF12] used the relaxed notion of *origin sessions*. However, both of these are still too restrictive for analyzing WireGuard, because this protocol does not explicitly authenticate the full transcript. Instead, for WireGuard, we are concerned matching only on a subset of the transcript information – the honest contributions of the keyshare and key-derivation materials. We introduce the notion of *contributive keyshares* to capture this intuition.

Definition 52 (Contributive keyshares)**.** Recall that $\pi_i^s.kid$ is the concatenation of all keyshare material sent by the session π_i^s during protocol execution. We say that π_j^t is a *contributive keyshare session* for π_i^s if $\pi_j^t.kid$ is a substring of $\pi_i^s.m_r$.

This definition is protocol specific: in WireGuard $\pi_i^s.kid$ consists only of the long-term public Diffie-Hellman value and the ephemeral public Diffie-Hellman value provided by the initiator and responder; in TLS 1.3 (for example) it would consist of the long-term public keys, the ephemeral public

Diffie-Hellman values and any pre-shared key identifiers provided by the client and selected by the server.

Cleanness Predicates

We now define the exact combinations of secrets that an adversary is allowed to leak without trivially breaking the protocol. The original cleanness predicate of Cremers and Feltz [CF12] allows the reveal of long-term secrets for the test session's party P_i at any time, which places us firmly in the setting where the adversary has key-compromise-impersonation abilities, but only allowed the reveal of long-term secrets of the intended peer after the test session has established a secure session, which captures perfect forward secrecy.

We now turn to modifying the cleanness predicate $\mathsf{clean}_{\mathsf{eCK\text{-}PFS\text{-}PSK}}$ for the pre-shared secret setting.

Definition 53 ($\mathsf{clean}_{\mathsf{eCK\text{-}PFS\text{-}PSK}}$)**.** A session π_i^s such that $\pi_i^s.\alpha = \texttt{accept}$ in the security experiment defined in Figure A.1 is $\mathsf{clean}_{\mathsf{eCK\text{-}PFS\text{-}PSK}}$ if all of the following conditions hold:

1. The query $\mathsf{Reveal}(i, s)$ has not been issued.
2. For all $(j, t) \in n_P \times n_S$ such that π_i^s *matches* π_j^t, the query $\mathsf{Reveal}(j, t)$ has not been issued.
3. If $\vec{\mathsf{PSKflag}}_i[\pi_i^s.pid] = \texttt{corrupt}$ or $\pi_i^s.psk = \bot$, the queries $\mathsf{CorruptASK}(i)$ and $\mathsf{CorruptEPK}(i, s)$ have not *both* been issued.
4. If $\vec{\mathsf{PSKflag}}_i[\pi_i^s.pid] = \texttt{corrupt}$ or $\pi_i^s.psk = \bot$, and for all $(j, t) \in n_P \times n_S$ such that π_j^t is a *contributive keyshare session* for π_i^s, then $\mathsf{CorruptASK}(j, t)$ and $\mathsf{CorruptEPK}(j, t)$ have not *both* been issued.
5. If there exists no $(j, t) \in n_P \times n_S$ such that π_j^t is a contributive keyshare session for π_i^s, $\mathsf{CorruptASK}(j)$ has not been issued before $\pi_i^s.\alpha \leftarrow \texttt{accept}$.

We specifically forbid the adversary from revealing the long-term and ephemeral secrets if the pre-shared secret between the test session and its intended partner has already been revealed. Since pre-shared keys are optional in our framework, we also must consider the scenario where a pre-shared secret does not exist between the test session π_i^s and its intended partner. Similarly, we forbid the adversary from revealing the long-term and ephemeral secrets if there exists no pre-shared secret between the two parties. Finally, since WireGuard does not authenticate the full transcript, but relies instead on implicit authentication of derived session keys based on secret information, we

must use our contributive keyshare partnering definition instead of the origin sessions of [CF12]. Like eCK-PFS, we capture perfect forward secrecy under key-compromise-impersonation attack in condition 5, where the long-term secret of the test session's intended partner is allowed to be revealed only after the test session has accepted. Additionally, we allow for the optional incorporation of pre-shared secrets in conditions 3 and 4, where the adversary falls back to eCK-PFS leakage paradigm if the pre-shared secret between the test session and its peer either does not already exist, or has been already revealed.

B. Post-Quantum Noise

B.1. The (Extended) Flexible ACCE Framework

In this section we give the full definition of the (extended) Flexible ACCE framework. We note that much of what follows is verbatim from the original fACCE model [DRS20], and that the differences between the two models (and a high-level overview of the fACCE model is given in Section 5.3.

B.1.1. fACCE Primitive Description

Recall that an fACCE protocol is a cryptographic protocol that establishes a secure channel between two parties. Eschewing a modular approach, channel establishment and payload transmission are handled by the same algorithms – where Send sends channel establishment information and payload data, and Recv receives. These functions may also update the internal state of the sessions. In addition to updating state and outputting ciphertext / plaintext (for Send and Recv, respectively) - these algorithms also output a stage flag ς. This flag ς can be used to indicate when an invocation of the algorithm reaches the next stage of security properties.

Definition 54 (Flexible ACCE)**.** A flexible ACCE protocol fACCE is a tuple of algorithms fACCE = (KGen, Init, Send, Recv) associated with a long-term secret key space $\mathcal{LSK}$, a long-term public key space $\mathcal{LPK}$, an ephemeral secret key space $\mathcal{ESK}$ an ephemeral public key space $\mathcal{EPK}$, and a state space $\mathcal{ST}$. The definition of fACCE algorithms are as follows:

- $\mathsf{KGen} \rightarrow_{\$} (sk, pk)$ generates a long-term keys where $sk \in \mathcal{LSK}, pk \in \mathcal{LPK}$.
- $\mathsf{Init}(sk, ppk, \rho, ad) \rightarrow_{\$} st$ initializes a session to begin communication, where sk (optionally) are the initiator's long-term secret keys, ppk (optionally) is the long-term public key of the intended session partner,

$\rho \in \{\mathtt{i}, \mathtt{r}\}$ is the session's role (i.e., initiator or responder), ad is data associated with this session, and $sk \in \mathcal{LSK} \cup \{\bot\}$, $ppk \in \mathcal{LPK} \cup \{\bot\}$, $ad \in \{0,1\}^*$, $st \in \mathcal{ST}$.

- $\mathsf{Send}(sk, st, m) \rightarrow_\$ (st', c)$ continues the protocol execution in a session and takes message m to output new state st', and ciphertext c, where $sk \in \mathcal{LSK} \cup \{\bot\}$, $st, st' \in \mathcal{ST}, m, c \in \{0,1\}^*$. Note that Send may generate additional ephemeral key pairs $(epk, esk) \in \mathcal{EPK} \times \mathcal{ESK}$.
- $\mathsf{Recv}(sk, st, c) \rightarrow_\$ (st', m)$ processes the protocol execution in a session triggered by c and outputs new state st', and message m, where $sk \in \mathcal{LSK} \cup \{\bot\}, st \in \mathcal{ST}, st' \in \mathcal{ST} \cup \{\bot\}, m, c \in \{0,1\}^*$. If $st' = \bot$ is output, then this denotes a rejection of this ciphertext.

As described in Section 5.3 we consider messages that are sent in an fACCE protocol to be sent in a ping-pong fashion, i.e. the initiator sends a message to the responder, which then replies with a message to the initiator. Multiple messages sent in a single flow are considered to be an extension of a single message. Each message sent monotonically increases the stage number of the protocol, i.e. stage one is the first message sent from the initiator to the responder, stage two is the first message sent from the responder to the initiator, etc. This is opposed to the original fACCE formulation, which only increased stage numbers upon achieving new security properties.

Furthermore, we only consider protocols with FIFO channels (i.e., protocols enforcing correct message order, and aborting for message omissions).

We define the correctness of an fACCE protocol below. Intuitively an fACCE protocol is correct if messages accepted from the established channel, were equally sent to this channel by the partner.

Definition 55 (Correctness of fACCE)**.** An fACCE protocol is correct if, for any two tuples $(sk_\mathtt{i}, pk_\mathtt{i}), (sk_\mathtt{r}, pk_\mathtt{r})$ output from KGen or set to $(\bot, \bot)$ respectively, their session states $\mathsf{Init}(sk_\mathtt{i}, pk_\mathtt{r}, \mathtt{i}, ad) \rightarrow_\$ st_\mathtt{i}, \mathsf{Init}(sk_\mathtt{r}, pk_\mathtt{i}, \mathtt{r}, ad) \rightarrow_\$ st_\mathtt{r}$ with $ad \in \{0,1\}^*$, and message-stage-ciphertext transcripts $MSC_\rho, MSC_{\bar{\rho}} \leftarrow \epsilon$, it holds for all sequences of operations $(op^0, \rho^0, m^0), \ldots, (op^n, \rho^n, m^n)$ (for all $0 \leq l \leq n$ with $op^l \in \{s, r\}, \rho^l \in \{\mathtt{i}, \mathtt{r}\}, m^l \in \{0,1\}^*$) that are executed as follows:

- if $op^l = s$, invoke $\mathsf{Send}(sk_{\rho^l}, st_{\rho^l}, m^l) \rightarrow_\$ (st_{\rho^l}, c^l)$ and update $MSC_\rho \leftarrow MSC_\rho \| (m^l, \varsigma^l, c^l)$, or
- if $op^l = r$, invoke $\mathsf{Recv}(sk_{\rho^l}, st_{\rho^l}, c^l) \rightarrow_\$ (st_{\rho^l}, m^l_*)$ on $(m^l_\circ, \varsigma^l_\circ, c^l) \| MSC_{\bar{\rho}} \leftarrow MSC_{\bar{\rho}}$ and update it accordingly,

that if $m_*^l \neq \perp$, then sent and received messages equal $m_*^l = m_\circ^l$.

B.1.2. Execution Environment

Here we describe the execution environment for our fACCE security experiment.

We consider a set of n_P parties each (potentially) maintaining a long-term key pair $\{(sk_1, pk_1), \ldots, (sk_{n_P}, pk_{n_P})\}$, $(sk_i, pk_i) \in \mathcal{LSK} \times \mathcal{LPK}$. Note that the long-term secret sk could contain both asymmetric secrets (such as long-term signing keys) and symmetric secrets (such as long-term preshared keys).

Each party can participate in up to n_S sessions, with each session potentially lasting n_T stages. Each session samples per-session randomness $rand$ used throughout the protocol execution. We denote both the set of variables that are specific for a session s of party i as well as the identifier of this session as π_i^s. In addition to the local variables specific to each protocol, we list the set of per-session variables that we require for our model below. In order to derive or modify a variable x of session π we write $\pi.x$ to specify this variable.

- $\rho \in \{\mathtt{i}, \mathtt{r}\}$: The role of the session in the protocol execution (i.e., initiator or responder).
- $\varsigma \in \mathbb{N}$: The current stage of the session.
- $pid \in [n_P]$: The session partner's identifier.
- ad: Data associated with this session (provided as parameter at session initialization to Init).
- $T_s[\cdot], T_r[\cdot] \in \{0,1\}^*$: Arrays of sent or received fACCE messages, which may consist of keying material, ciphertexts or even plaintexts[1]. After every invocation of Send or Recv of a session π_i^s, the respective ciphertext is appended to $\pi_i^s.T_s$ or $\pi_i^s.T_r$ respectively.
- $st \in \mathcal{ST}$: All protocol-specific local variables.
- $rand \in \{0,1\}^*$: Any random coins used by π_i^s's protocol execution.
- $\pi_i^s.rr \in \{0,1\}$: A flag indicating if $\mathcal{A}$ has leaked the ephemeral randomness used during the session execution.

[1] Note that in what follows, we refer to these messages generically as "ciphertexts".

- $(b_1, b_2, b_3, ..., b_{n_T})$: A vector of challenge bits the adversary is to guess (one bit for each stage).
- $fr_1, fr_2, ..., fr_{n_T} \in \{0,1\}$: Freshness flags for the security game, indicating whether the adversary has caused the challenge bit to be trivial to guess.
- FT: A modifiable copy of the fACCE protocol's security table ST for the session π_i^s.

At the beginning of the game, for all sessions π_i^s the following initial values are set: $\pi_i^s.T_s$, $\pi_i^s.T_r$, $\leftarrow \epsilon$, and $\pi_i^s.\mathsf{FT} \leftarrow \mathsf{ST}$, and $\pi_i^s.fr_{\varsigma^*} \leftarrow 1$ for all $\varsigma^* \in [0, ..., n_T]$, and $\pi_i^s.rand \xleftarrow{\$} \{0,1\}^*$, $\pi_i^s.b_{\varsigma^*} \xleftarrow{\$} \{0,1\}$ for all $\varsigma^* \in \{1, ..., n_T\}$ are sampled.

Furthermore a set of ciphertexts $Rpl \leftarrow \emptyset$ is maintained in the security game, that are declared to initiate a non-fresh (replayed) session.

Partnering In order to define security in a flexible manner, we need to define partnering for sessions in the environment. Partnering is defined over the ciphertexts provided to/by the adversary via the oracles that let sessions send and receive ciphertexts ($\mathsf{OSend}, \mathsf{ORecv}$). Intuitively, a session has an honest partner if everything that the honest partner received via ORecv was sent by the session via OSend (without modification) and vice versa, and at least one of the two parties received a ciphertext at least once. This definition considers the asynchronous nature of the established channel, leading to a *matching conversation*-like partnering definition for fACCE.

Definition 56 (Honest Partner)**.** π_j^t is an honest partner of π_i^s if all initial variables match ($\pi_i^s.pid = j$, $\pi_j^t.pid = i$, $\pi_i^s.\rho \neq \pi_j^t.\rho$, $\pi_i^s.ad = \pi_j^t.ad$) and the received transcripts are a prefix of the partner's sent transcripts, where at least one them is not empty (i.e., for $a = |\pi_j^t.T_r|$, $b = |\pi_i^s.T_r|$ such that $a > 0$ if $\pi_i^s.\rho = \mathtt{i}$ and $b > 0$ if $\pi_i^s.\rho = \mathtt{r}$ then $\forall\ 0 \leq \alpha < a : (\pi_i^s.T_s[\alpha] = \pi_j^t.T_r[\alpha])$ and $\forall\ 0 \leq \beta < b : (\pi_i^s.T_r[\beta] = \pi_j^t.T_s[\beta])$). If π_i^s already received ciphertexts from π_j^t, then π_j^t is an honest partner of π_i^s only if there exists no other honest partner π^* of π_i^s (i.e., if $b > 0$ then there is no π^* such that π^* is an honest partner of π_i^s and $\pi^* \neq \pi_j^t$).

Please note that after sending a message that has not yet been received, the initiator may have multiple honest partners (if the resulting ciphertexts are forwarded to multiple sessions). Due to the last requirement in Definition 56, our partnering notion requires that, after decrypting once, a session must have

no more than one honest partner. Thereby partnering necessarily becomes a 1-to-1 relation as soon as the initiator received once from the responder.

Additional Partner Notion The reveal of ephemeral randomness of a session does not only affect current honest partners (see Definition 56) but also sessions that previously were honest partners of the session for which the randomness was revealed. Thus we must define *Previous Honest Partner*:

Definition 57 (Previous Honest Partner)**.** We say that π_j^t is a previous honest partner of π_i^s if $\pi_i^s.pid = j$, $\pi_j^t.pid = i$, $\pi_i^s.\rho = \pi_j^t.\bar{\rho}$, $\pi_i^s.ad = \pi_j^t.ad$, $\pi_i^s.T_r$ and $\pi_j^t.T_s$ have a common prefix, and $\pi_j^t.T_r$ and $\pi_i^s.T_s$ have a common prefix where at least one prefix is not empty (i.e., for $a \leq |\pi_j^t.T_r|$, $b \leq |\pi_i^s.T_r|$ such that $a > 0$ if $\pi_i^s.\rho = \mathtt{i}$ and $b > 0$ if $\pi_i^s.\rho = \mathtt{r}$ then $\forall\ 0 \leq \alpha < a : (\pi_i^s.T_s[\alpha] = \pi_j^t.T_r[\alpha]) \wedge \forall 0\ \leq \beta < b : (\pi_i^s.T_r[\beta] = \pi_j^t.T_s[\beta]))$.

The main differences towards an *honest partner* are that: (a) In *previous honest partners* a and b can be less than or equal $|\pi_j^t.T_r|$ and $|\pi_i^s.T_r|$ respectively (meaning that π_i^s and π_j^t were honest partners once) and due to this; (b) It is not (and actually cannot be) required that there exists only one *previous honest partner*.

B.1.3. Flexible Security Notion

To facilitate the security game, the challenger maintains for each session π_i^s a set $\mathbb{S}_{\pi_i^s}$ that contains labels of all secrets that each session (and its honest partner) maintains – the long-term secret values sk_i, sk_j (both asymmetric and symmetric), all ephemeral secret values sampled during the n_T stages of the protocol execution $esk_s^1, esk_t^1 \ldots, esk_s^{n_T}, esk_t^{n_T}$ and the state maintained during the protocol executions at each stage $st_s^1, st_t^1 \ldots, st_s^{n_T}, st_t^{n_T}$. Thus $\mathbb{S}_{\pi_i^s}$ $= (sk_i, sk_j, esk_s^1, esk_t^1, \ldots, esk_s^{n_T}, esk_s^{n_T}, st_s^1, st_t^1 \ldots, st_s^{n_T}, st_t^{n_T})$.

This $\mathbb{S}_{\pi_i^s}$ will be used to generate the security table ST used to determine the security of any session of the associated protocol under certain compromise patterns. Note that this corresponds to the counters maintained in previous versions of the fACCE framework - each counter corresponded to a particular compromise query allowed under a certain security property, and stated the stage in which security would be reached under this compromise pattern.

Each fACCE protocol is associated with a security table ST, which will be copied into a freshness table FT for each session π_i^s in the protocol execution. ST is made up of 4 column rows $(\vec{s}, \mathtt{cf}, \mathtt{au}^{\mathtt{i}}, \mathtt{au}^{\mathtt{r}})$ where $\vec{s}$ is a vector of some

secrets maintained by the challenger during the execution of the fACCE security game corresponding to some element in the powerset $\mathbb{S}_{\pi_i^s}$, and $\mathtt{cf}$, $\mathtt{au}^{\mathtt{i}}$, and $\mathtt{au}^{\mathtt{r}}$ are stage counters. The intuition here is that each row describes what stages confidentiality ($\mathtt{cf}$), authentication of initiators ($\mathtt{au}^{\mathtt{i}}$) and authentication of responders ($\mathtt{au}^{\mathtt{r}}$) is achieved when the collection of secrets $\vec{s}$ remains uncompromised by the attacker.

Replay Attacks The previously introduced partnering notion already defines session participants unpartnered for all but one type of replay attacks: if ciphertexts, sent by an initiator that has already received a ciphertext once, or sent by a responder, are replayed, the respective receiver is defined to have no honest partner. In a security game in which state reveals are defined to be harmless for unpartnered sessions (which is the case for our model), this induces that such replay attacks force the protocol to diverge respective receivers' session states from their previous partners' session states. As a consequence, only replays of ciphertexts, sent by an initiator to (multiple) responder(s) without any reply from the latter, must be considered harmful in our security experiment. These replay attacks cannot be prevented if the receiver's long-term secret is defined static and the initiator has never received a ciphertext. Our definition of replay attack resistance consequently focuses on the security damage that is caused by such replay attacks: it considers how soon the secrets, established by a (replayed) ciphertext, are independent among the sender and the (other) receivers of this replayed ciphertext. Hence, a session's secrets are recovered from a replay attack if they cannot be used to obtain information on other sessions' secrets.

Besides the explained prevention of replay attacks due to our partnering notion, ciphertexts that are transmitted before a stage $\varsigma > 0$ is output are (as also explained above) authenticated as soon as authentication is reached in a later stage. Apart from this, no security guarantees are required for ciphertexts transmitted under $\varsigma = 0$.

If a property is never reached in the specified protocol, then the respective counter is set to ∞ (e.g., for protocol with unauthenticated initiators, $\mathtt{au}^{\mathtt{i}} = \infty$).

B.1.4. Adversarial Model

In order to model active attacks in our environment, the security experiment provides the OInit, OSend, ORecv oracles to an adversary $\mathcal{A}$, who can use them to control communication among sessions, together with the oracles $\mathsf{OCorrupt}$, $\mathsf{OReveal}$ and $\mathsf{ORevealRandomness}$.

Following the direction of the original fACCE work, we treat the authentication and confidentiality properties similarly to the original AEAD notion of Rogaway [Rog02]: the game maintains a win flag (to indicate whether the adversary broke authenticity or integrity of ciphertexts) and embeds challenge bits in the encryption (in order to model indistinguishability of ciphertexts). In order to win the security game, adversary $\mathcal{A}$ either has to trigger $\mathsf{win} \leftarrow 1$ or output the correct challenge bit $\pi_i^s.b_\varsigma$ of a specific session stage ς at the end of the game.

In addition, the Challenger maintains a set of freshness flags $\pi_i^s.fr_\varsigma$ for each stage ς of each session π_i^s. When $\mathcal{A}$ makes a query to $\mathsf{OCorrupt}$, $\mathsf{OReveal}$ or $\mathsf{ORevealRandomness}$, then $\mathcal{C}$ deletes all rows $(\vec{s}_j, \mathtt{cf}_j, \mathtt{au}_j^{\mathtt{i}}, \mathtt{au}_j^{\mathtt{r}})$ in $\pi_i^s.\mathsf{FT}$ if the associated secret (esk_s^t, sk_i, etc.) of the query is in the row, i.e. $esk_s^t \in \vec{s}_j$. All stages for all sessions that are not an element of the right-hand columns are now considered un-fresh, and the corresponding freshness flags are set to 0. In particular, if there does not exist a counter $\mathtt{cf}_j$ such that $\varsigma = \mathtt{cf}_j$ where $(\vec{s}_j, \mathtt{cf}_j, \mathtt{au}_j^{\mathtt{i}}, \mathtt{au}_j^{\mathtt{r}} \in \pi_i^s.\mathsf{FT})$ for all $j \in \{0, ... |\mathsf{FT}|\}$ [2] , then $\pi_i^s.\varsigma \leftarrow 0$. When $\mathcal{A}$ terminates and outputs a session π_i^s and a stage counter ς, if the freshness flag associated with $\pi_i^s.\varsigma$ is 0, then $\mathcal{C}$ simply outputs a random bit b^* instead of $\pi_i^s.b_\varsigma = b'$.

- $\mathsf{OInit}(i, pk_j, \rho, ad)$ initializes a new session π_i^s of party i to be partnered with party j, invoking $\text{fACCE.}\mathsf{Init}(sk_i, pk_j, \rho, ad) \rightarrow_{[\pi_i^s.rand]} \pi_i^s.st$ under $\pi_i^s.rand$. It also sets $\pi_i^s.\rho \leftarrow \rho$, $\pi_i^s.pid \leftarrow j$, and $\pi_i^s.ad \leftarrow ad$. This oracle provides the new session index s. All subsequent invocations of $\mathsf{Send}, \mathsf{Recv}$ of this session participant use $\pi_i^s.rand$ for obtaining randomness.

- $\mathsf{OSend}(i, s, m_0, m_1)$ triggers the encryption of a message m_b for $b = \pi_i^s.b_\varsigma$ by invoking $\mathsf{Send}(sk_i, \pi_i^s.st, m_b) \rightarrow_{[\pi_i^s.rand]} (st', c)$ for an initialized π_i^s if $|m_0| = |m_1|$ and returns $\perp$ otherwise. It updates the session specific variables $\pi_i^s.st \leftarrow st'$, returns $(c, \pi_i^s.\varsigma)$ to the adversary, and appends c to $\pi_i^s.T_s$ if $c \neq \perp$. If c is the first ciphertext and $\pi_i^s.\rho = \mathtt{i}$, then $Rpl \leftarrow Rpl \cup \{c\}$. Note that c contains both the explicit ciphertext encrypting the message m_b and any channel establishment messages that are send in this stage.

- $\mathsf{ORecv}(i, s, c)$ triggers invocation of $\mathsf{Recv}(sk_i, \pi_i^s.st, c) \rightarrow_{[\pi_i^s.rand]} (st', m)$ for an initialized π_i^s and returns $(m, \pi_i^s.\varsigma)$ if π_i^s has no honest partner

[2] For conciseness, we will say for all $j \in \pi_i^s.[|\mathsf{FT}|]$ as shorthand for "for all $(\vec{s}_j, \mathtt{cf}_j, \mathtt{au}_j^{\mathtt{i}}, \mathtt{au}_j^{\mathtt{r}}) \in \pi_i^s.\mathsf{FT}$, for all $j \in \{0, ... |\mathsf{FT}|\}$."

(since challenges from the encryption oracle would otherwise be trivially leaked), or returns $\pi_i^s.\varsigma$ otherwise. Finally c is appended to $\pi_i^s.T_r$ if decryption succeeds.

Excluding trivial attacks:

Conf: Since decryption can change the honesty of partners, the freshness flags are updated regarding corruptions and the reveal of ephemeral randomness.

Auth: If the received ciphertext was not sent by a session of the intended partner (i.e., there exists no honest partner) and authentication of the partner

1. was not reached yet (i.e., if $\varsigma \neq \mathtt{au}_j^{\pi_i^s.\bar{\rho}}$ for all $j \in \pi_i^s.[|\mathsf{FT}|]$), then all following stages are marked un-fresh until authentication will be reached ($\pi_i^s.fr_{\varsigma^*} \leftarrow 0$ for all $\varsigma \leq \varsigma^* < \mathsf{min}(\mathtt{au}_j^{\pi_i^s.\bar{\rho}})$ for all $\mathtt{au}_j^{\pi_i^s.\bar{\rho}} \in \pi_i^s.\mathsf{FT}$), since this is a (temporarily) trivial impersonation of the partner towards π_i^s.
2. was reached before, but $\mathcal{A}$ has since made a query that would allow trivial impersonation of the partner towards π_i^s. In that case, there would exist no $\mathtt{au}_j^{\pi_i^s.\bar{\rho}} \leq \varsigma$ for all $j \in \pi_i^s.[|\mathsf{FT}|]$ (i.e., $\mathcal{A}$ has made a query to allow trivial impersonation again, which deleted rows from $\pi_i^s.\mathsf{FT}$), then all following stages are marked un-fresh ($\pi_i^s.fr_{\varsigma^*} \leftarrow 0$ for all $\varsigma \leq \mathsf{min}(\mathtt{au}_j^{\pi_i^s.\bar{\rho}})$), since this is a trivial impersonation of the partner towards π_i^s.

Rewarding real attacks:

Auth: Similarly to detecting trivial attacks, real attacks are rewarded by considering the goals that are defined to be reached by the protocol and the corruptions of the participants' long term secrets.

The adversary breaks authentication (and thereby $\mathsf{win} \leftarrow 1$ is set) if the received ciphertext was not sent by a session of the intended partner but was successfully received (i.e., there exists no honest partner and the output state is $st' \neq \bot$), the stage is still fresh ($\pi_i^s.fr_\varsigma = 1$), and $\mathcal{A}$ has not issued queries that trivially break authentication, i.e. there exists $\mathtt{au}_j^{\pi_i^s.\bar{\rho}} \in \mathsf{ST}$ such that $\varsigma = \mathtt{au}_j^{\pi_i^s.\bar{\rho}}$.

- $\mathsf{ORevealRandomness}(i, s) \rightarrow rand$ outputs the randomness $\pi_i^s.rand$ sampled by party i in its session π_i^s. The freshness table $\pi_i^s.\mathsf{FT}$ is updated in the following way: for all $j \in \pi_i^s.[|\mathsf{FT}|]$ if $rand \in \vec{s}_j$ where $(\vec{s}_j, \mathtt{cf}_j, \mathtt{au}_j^{\mathtt{i}}, \mathtt{au}_j^{\mathtt{r}}) \in \pi_i^s.\mathsf{FT}$, then $\pi_i^s.\mathsf{FT} \leftarrow \pi_i^s.\mathsf{FT} \setminus \{(\vec{s}_j, \mathtt{cf}_j, \mathtt{au}_j^{\mathtt{i}}, \mathtt{au}_j^{\mathtt{r}})\}$.

Freshness flags are updated similarly: for all $j \in \pi_i^s.[|\mathsf{FT}|]$ if $\nexists\ \mathtt{cf}_j$ such that $\varsigma \neq \mathtt{cf}_j$ where $(\vec{s}_j, \mathtt{cf}_j, \mathtt{au}_j^{\mathtt{i}}, \mathtt{au}_j^{\mathtt{r}}) \in \pi_i^s.\mathsf{FT}$, then $\pi_i^s.fr_\varsigma \leftarrow 0$.

- $\mathsf{OCorrupt}(i) \rightarrow sk_i$ outputs the long-term secret keys sk_i of party i, and sets $corr_i \leftarrow 1$. The freshness table $\pi_i^s.\mathsf{FT}$ is updated in the following way: for all $j \in \pi_i^s.[|\mathsf{FT}|]$ if $sk_i \in \vec{s}_j$ where $(\vec{s}_j, \mathtt{cf}_j, \mathtt{au}_j^{\mathtt{i}}, \mathtt{au}_j^{\mathtt{r}}) \in \pi_i^s.\mathsf{FT}$, then $\pi_i^s.\mathsf{FT} \leftarrow \pi_i^s.\mathsf{FT} \setminus \{(\vec{s}_j, \mathtt{cf}_j, \mathtt{au}_j^{\mathtt{i}}, \mathtt{au}_j^{\mathtt{r}})\}$. Freshness flags are updated similarly: for all $j \in \pi_i^s.[|\mathsf{FT}|]$ if $\nexists\ \mathtt{cf}_j$ such that $\varsigma \neq \mathtt{cf}_j$ where $(\vec{s}_j, \mathtt{cf}_j, \mathtt{au}_j^{\mathtt{i}}, \mathtt{au}_j^{\mathtt{r}}) \in \pi_i^s.\mathsf{FT}$, then $\pi_i^s.fr_\varsigma \leftarrow 0$.

- $\mathsf{OReveal}(i, s) \rightarrow \pi_i^s.st$ outputs the current session state $\pi_i^s.st$. The freshness table $\pi_i^s.\mathsf{FT}$ is updated in the following way: for all $j \in |\pi_i^s.\mathsf{FT}|$ if $\pi_i^s.st \in \vec{s}_j$ where $(\vec{s}_j, \mathtt{cf}_j, \mathtt{au}_j^{\mathtt{i}}, \mathtt{au}_j^{\mathtt{r}}) \in \pi_i^s.\mathsf{FT}$, then $\pi_i^s.\mathsf{FT} \leftarrow \pi_i^s.\mathsf{FT} \setminus \{(\vec{s}_j, \mathtt{cf}_j, \mathtt{au}_j^{\mathtt{i}}, \mathtt{au}_j^{\mathtt{r}})\}$. Freshness flags are updated similarly: for all $j \in |\pi_i^s.\mathsf{FT}|$ if $\nexists\ \mathtt{cf}_j$ such that $\varsigma \neq \mathtt{cf}_j$ where $(\vec{s}_j, \mathtt{cf}_j, \mathtt{au}_j^{\mathtt{i}}, \mathtt{au}_j^{\mathtt{r}}) \in \pi_i^s.\mathsf{FT}$, then $\pi_i^s.fr_\varsigma \leftarrow 0$.
  Excluding trivial attacks:
  - Revealing the session-state trivially determines this session's challenge bits, since the state contains any used session keys[3]. Hence $\pi_i^s.fr_{\varsigma^*} \leftarrow 0$ is set for all stages ς^*.
  - Similarly, sufficient information is leaked to determine challenge bits embedded in ciphertexts to and from *all* honest partners π_j^t (and to impersonate π_i^s towards them). As such, for all sessions π_j^t such that π_j^t is an honest partner or previous honest partner of π_i^s, $\pi_j^t.fr_{\varsigma^*} \leftarrow 0$ is set for all stages ς^*.
  - In case the revealed secrets enable the adversary to obtain secrets of non-partnered sessions due to a replay attack then the first ciphertext in this session is declared to induce non-fresh sessions via $Rpl \leftarrow Rpl \cup \{c\}$ where $c \leftarrow \pi_i^s.T_s[0]$ if $\pi_i^s.\rho = \mathtt{i}$ or $c \leftarrow \pi_i^s.T_r[0]$ if $\pi_i^s.\rho = \mathtt{r}$ (such that all sessions starting with the same ciphertext are also marked non-fresh). Afterwards, for all sessions π_j^t such that $\pi_j^t.\rho = \mathtt{r}$ (respectively $\pi_j^t.\rho = \mathtt{i}$), $c = \pi_j^t.T_r[0]$ (respectively $c = \pi_j^t.T_s[0]$) and $c \in Rpl$, we set $\pi_j^t.fr_\varsigma \leftarrow 0$ for all ς.

[3] Since we do not consider forward-secrecy within sessions, the secret session state is considered to harm security of the whole session lifetime independent of when the state is revealed.

B.1.5. Security Definition

Definition 58 (Advantage in Breaking Flexible ACCE)**.** An adversary $\mathcal{A}$ breaks a flexible ACCE protocol fACCE with security table ST, capturing authentication, key compromise impersonation resilience, forward-secrecy, eCK-security, and replayability resistance, when $\mathcal{A}$ plays the fACCE game, and outputs $\mathsf{Exp}^{\mathrm{fACCE},\mathsf{ST}}_{n_P,n_S,\mathcal{A}}(\lambda) = 1$. We define the advantage of an adversary $\mathcal{A}$ breaking a flexible ACCE protocol fACCE as $\mathsf{Adv}^{\mathrm{fACCE},\mathsf{ST}}_{\mathcal{A}} = \Pr[\mathsf{Exp}^{\mathrm{fACCE},\mathsf{ST}}_{n_P,n_S,\mathcal{A}}(\lambda) = 1]$.

Intuitively, a flexible ACCE protocol fACCE is secure if it is correct and $\mathsf{Adv}^{\mathrm{fACCE},\mathsf{ST}}_{\mathcal{A}}$ is negligible for all probabilistic algorithms $\mathcal{A}$ running in polynomial-time. A flexible ACCE protocol fACCE is post-quantum secure if it is correct and $\mathsf{Adv}^{\mathrm{fACCE},\mathsf{ST}}_{\mathcal{Q}}$ is negligible for all quantum algorithms $\mathcal{Q}$ running in polynomial-time.

B.2. PRP-SEEC

To provide a simple and efficient instantiation for SEEC that is both practical and demonstrates the feasibility of our security-notion for SEEC, we introduce PRP-SEEC, defined in Algorithm 12. It can be summarized as combining a random static key with the ephemeral entropy via a pseudo random permutation PRP (as defined in Appendix 2.2.6), where it uses the static key as key and the ephemeral entropy as message for the PRP. The advantage of this approach is that it achieves information-theoretical security in case of uncompromised ephemeral randomness and may be able to use existing hardware-acceleration for specific PRP-schemes, such as AES.

Theorem 28. *A PRP-SEEC-scheme Σ is a secure SEEC-scheme with:*

$$\mathsf{Adv}^{SEEC}_{\Sigma,\mathcal{A}}(1^\lambda) \leq \frac{2n^2}{2^\lambda} + \mathsf{Adv}^{\mathit{IND\text{-}CPA}}_{PRP,\mathcal{A}'}(1^\lambda),$$

where n is the number of $\mathsf{GenRand}$*-queries that $\mathcal{A}$ performs.*

Proof. $\mathcal{A}$ can only win if he does not reveal both the static key and the randomness of the challenge-session. We can thus distinguish the case in which he does not receive the challenge-randomness and the case in which he does not receive the static key.

In the first case the challenge-randomness is random and independent and used as input to a permutation. Applying a permutation to a random and

Experiment 19: The fACCE experiment for adversary $\mathcal{A}$. Note that for readability, when context is clear we use b as shorthand for $\pi_i^s.b_{\pi_i^s.\varsigma}$. Part and $\mathsf{PrevPart}$ are functions that capture *Honest Partnering* and *Previous Honest Partnering* definitions, respectively. Finally, comb is a variable that captures whether the protocol does randomness hardening by combining ephemeral randomness with long-term secret information. The missing oracles are given in Experiments 20–24.

$\mathsf{win} \leftarrow 0, Rpl \leftarrow \emptyset$
for $i \in \{1, ..., n_P\}$**:**
  $ct_i \leftarrow 1$
  $corr_i \leftarrow 0$
  $(pk_i, sk_i) \xleftarrow{\$} \mathsf{KGen}(\lambda)$
$(i, s, \varsigma, b') \xleftarrow{\$} \mathcal{A}^{\mathsf{OInit},\mathsf{OSend},\mathsf{ORecv},\mathsf{OReveal},\mathsf{ORevealRandomness},\mathsf{OCorrupt}}(pk_1, \dots, pk_{n_P})$
if $\mathsf{win} = 1$**:**
  return 1
if $\pi_i^s.fr_\varsigma = 0$**:**
  return $b^* \xleftarrow{\$} \{0, 1\}$
else:
  return $(b' = \pi_i^s.b_\varsigma)$
Oracle $\mathsf{OSend}(i, s, m_0, m_1) \rightarrow (c, \varsigma)$**:**
  if $|m_0| \neq |m_1|$**:**
    return $\perp$
  $(st', c) \xleftarrow{\$} \text{fACCE.}\mathsf{Send}(sk_i, \pi_i^s.st, m_b)$
  $\pi_i^s.st \leftarrow st'$
  if $(c \neq \perp)$**:**
    if $(\pi_i^s.T_s = \epsilon) \wedge (\pi_i^s.\rho = \mathtt{i})$**:**
      $Rpl \leftarrow Rpl \cup \{c\}$
  $\pi_i^s.T_s \leftarrow \pi_i^s.T_s || c$
  return $c, \pi_i^s.\varsigma$

Experiment 20: The ORecv-oracle of the fACCE-game.

Oracle $\mathsf{ORecv}(i, s, c) \to (\varsigma)$**:**
- $(st', m) \xleftarrow{\$} \text{fACCE.}\mathsf{Recv}(sk_i, \pi_i^s.st, c)$
- $\pi_i^s.st \leftarrow st'$
- **if** $m \neq \perp$**:**
  - $\pi_i^s.T_r \leftarrow \pi_i^s.T_r || c$
  - **if** $(\exists (j,t) : \mathsf{Part}(\pi_i^s, \pi_j^t) = 1)$**:**
    - **if** $(\pi_j^t.rr = 1) \wedge (\mathsf{comb} = 0)$**:**
      - **for** $u \in \{1, ..., |\pi_i^s.\mathsf{FT}|\}$**:**
        - $(\vec{s}_u, \mathtt{cf}_u, \mathtt{au}_u^{\mathtt{i}}, \mathtt{au}_u^{\mathtt{r}}) \leftarrow \pi_i^s.\mathsf{FT}[u]$
        - **if** $(esk_j \in \vec{s}_u)$**:**
          - $\pi_i^s.\mathsf{FT}[u] \leftarrow \emptyset$
    - **if** $(\pi_j^t.rr = 1) \wedge (corr_j) \wedge (\mathsf{comb} = 1)$**:**
      - **for** $u \in \{1, ..., |\pi_i^s.\mathsf{FT}|\}$**:**
        - $(\vec{s}_u, \mathtt{cf}_u, \mathtt{au}_u^{\mathtt{i}}, \mathtt{au}_u^{\mathtt{r}}) \leftarrow \pi_i^s.\mathsf{FT}[u]$
        - **if** $(esk_j \in \vec{s}_u)$**:**
          - $\pi_i^s.\mathsf{FT}[u] \leftarrow \emptyset$
    - **for** $\varsigma \in \{1, ..., n_T\}$**:**
      - **if** $(\nexists\ \mathtt{cf}_u \in \pi_i^s.\mathsf{FT} : \varsigma \geq \mathtt{cf}_u)$**:**
        - $\pi_i^s.fr_\varsigma \leftarrow 0.$
    - **if** $(\nexists (j,t) : \pi_j^t.T_s \supseteq \pi_i^s.T_r)$**:**
      - **if** $(\nexists \mathtt{au}_u^{\pi_i^s.\hat{\rho}} \in \pi_i^s.\mathsf{FT} : \mathtt{au}_u^{\pi_i^s.\hat{\rho}} \leq \pi_i^s.\varsigma) \wedge (c \notin Rpl)$**:**
        - $\mathsf{win} \leftarrow 1$
    - $\pi_i^s.\varsigma{+}{+}$
- **return** $\pi_i^s.\varsigma$

Experiment 21: The OInit-oracle of the fACCE-game.

Oracle $\mathsf{OInit}(i, pk_j, \rho, ad) \to s$**:**
  if $(pk_j \notin \{pk_1, ... pk_{n_P}\})$**:**
    return $\perp$
  $s \leftarrow ct_i,\ ct_i{+}{+}$
  $\pi_i^s.\rho \leftarrow \rho,\ \pi_i^s.T_s, \pi_i^s.T_r \leftarrow \epsilon$
  $\pi_i^s.pid \leftarrow j,\ \pi_i^s.ad \leftarrow ad$
  for $\varsigma \in \{1, ..., n_T\}$**:**
    $\pi_i^s.fr_\varsigma \leftarrow 1$
    $\pi_i^s.rand \xleftarrow{\$} \{0,1\}^*$
    $\pi_i^s.rr \leftarrow 0$
    $\pi_i^s.b_\varsigma \xleftarrow{\$} \{0,1\}$
  $\pi_i^s.st \xleftarrow{\$} \text{fACCE.}\mathsf{Init}(sk_i, pk_j, \rho, ad)$
  $\pi_i^s.\mathsf{FT} \leftarrow \mathsf{ST}$
  if $corr_{\pi_i^s.pid} = 1$**:**
    for $u \in \{1, ..., |\pi_i^s.\mathsf{FT}|\}$**:**
      $(\vec{s}_u, \mathtt{cf}_u, \mathtt{au}_u^{\mathtt{i}}, \mathtt{au}_u^{\mathtt{r}}) \leftarrow \pi_i^s.\mathsf{FT}[u]$
      if $(sk_j \in \vec{s}_u)$**:**
        $\pi_i^s.\mathsf{FT}[u] \leftarrow \emptyset$
  for $\varsigma \in \{1, ..., n_T\}$**:**
    if $(\nexists\ \mathtt{cf}_u \in \pi_i^s.\mathsf{FT} : \varsigma \geq \mathtt{cf}_u)$**:**
      $\pi_j^t.fr_\varsigma \leftarrow 0.$
  return s

Experiment 22: The $\mathsf{OCorrupt}$-oracle of the fACCE-game.

Oracle $\mathsf{OCorrupt}(i) \rightarrow (sk_i)$**:**
  $corr_i \leftarrow 1$ **for** $s \in \{1, ..., n_S\}$**:**
    for $u \in \{1, ..., |\pi_i^s.\mathsf{FT}|\}$**:**
      $(\vec{s}_u, \mathtt{cf}_u, \mathtt{au}_j^{\mathtt{i}}, \mathtt{au}_j^{\mathtt{r}}) \leftarrow \pi_i^s.\mathsf{FT}[u]$ **if** $(sk_i \in \vec{s}_u)$**:**
        $\pi_i^s.\mathsf{FT}[u] \leftarrow \emptyset$
  for $j \in \{1, ..., n_P\}$**:**
    for $t \in \{1, ..., n_S\}$**:**
      if $(\pi_j^t.pid = i)$**:**
        for $u \in \{1, ..., |\pi_j^t.\mathsf{FT}|\}$**:**
          $(\vec{s}_u, \mathtt{cf}_u, \mathtt{au}_j^{\mathtt{i}}, \mathtt{au}_j^{\mathtt{r}}) \leftarrow \pi_j^t.\mathsf{FT}[u]$
          if $(sk_i \in \vec{s}_u)$**:**
            $\pi_j^t.\mathsf{FT}[u] \leftarrow \emptyset$
        for $\varsigma \in \{1, ..., n_T\}$**:**
          if $(\nexists\ \mathtt{cf}_u \in \pi_j^t.\mathsf{FT} : \varsigma \geq \mathtt{cf}_u)$**:**
            $\pi_j^t.fr_\varsigma \leftarrow 0$
  for $\varsigma \in \{1, ..., n_T\}$**:**
    if $(\nexists\ \mathtt{cf}_u \in \pi_i^s.\mathsf{FT} : \varsigma \geq \mathtt{cf}_u)$**:**
      $\pi_i^s.fr_\varsigma \leftarrow 0.$
  if $(\pi_i^s.rr = 1) \wedge (\mathsf{comb} = 1)$**:**
    for $u \in \{1, ..., |\pi_i^s.\mathsf{FT}|\}$**:**
      $(\vec{s}_u, \mathtt{cf}_u, \mathtt{au}_u^{\mathtt{i}}, \mathtt{au}_u^{\mathtt{r}}) \leftarrow \pi_i^s.\mathsf{FT}[u]$
      if $(esk_j \in \vec{s}_u)$**:**
        $\pi_i^s.\mathsf{FT}[u] \leftarrow \emptyset$
  return sk_i

Experiment 23: The $\mathsf{OReveal}$-oracle of the fACCE-game.

Oracle $\mathsf{OReveal}(i, s) \to \pi_i^s.st$**:**
  for $u \in \{1, ..., |\pi_i^s.\mathsf{FT}|\}$**:**
    $(\vec{s}_u, \mathtt{cf}_u, \mathtt{au}_j^{\mathtt{i}}, \mathtt{au}_j^{\mathtt{r}}) \leftarrow \pi_i^s.\mathsf{FT}[u]$
    if $(\pi_i^s.st \in \vec{s}_u)$**:**
      $\pi_i^s.\mathsf{FT}[u] \leftarrow \emptyset$
  for $j \in \{1, ..., n_P\}$**:**
    for $t \in \{1, ..., n_S\}$**:**
      if $(\mathsf{PrevPart}(\pi_i^s, \pi_j^t))$**:**
        for $u \in \{1, ..., |\pi_j^t.\mathsf{FT}|\}$**:**
          $(\vec{s}_u, \mathtt{cf}_u, \mathtt{au}_j^{\mathtt{i}}, \mathtt{au}_j^{\mathtt{r}}) \leftarrow \pi_j^t.\mathsf{FT}[u]$
          if $(\pi_i^s.st \in \vec{s}_u)$**:**
            $\pi_i^s.\mathsf{FT}[u] \leftarrow \emptyset$
        for $\varsigma \in \{1, ..., n_T\}$**:**
          if $(\nexists\ \mathtt{cf}_u \in \pi_j^t.\mathsf{FT} : \varsigma \geq \mathtt{cf}_u)$**:**
            $\pi_j^t.fr_\varsigma \leftarrow 0.$
  return $\pi_i^s.st$

Experiment 24: The $\mathsf{ORevealRandomness}$-oracle of the fACCE-game.

Oracle $\mathsf{ORevealRandomness}(i, s, \varsigma) \rightarrow \pi_i^s.rand$**:**
- $\pi_i^s.rr \leftarrow 1$
- **if** $(\mathsf{comb} = 0)$**:**
  - **for** $u \in \{1, ..., |\pi_i^s.\mathsf{FT}|\}$**:**
    - $(\vec{s}_u, \mathtt{cf}_u, \mathtt{au}_u^{\mathtt{i}}, \mathtt{au}_u^{\mathtt{r}}) \leftarrow \pi_i^s.\mathsf{FT}[u]$
    - **if** $(esk_j \in \vec{s}_u)$**:**
      - $\pi_i^s.\mathsf{FT}[u] \leftarrow \emptyset$
- **if** $(corr_i) \wedge (\mathsf{comb} = 1)$**:**
  - **for** $u \in \{1, ..., |\pi_i^s.\mathsf{FT}|\}$**:**
    - $(\vec{s}_u, \mathtt{cf}_u, \mathtt{au}_u^{\mathtt{i}}, \mathtt{au}_u^{\mathtt{r}}) \leftarrow \pi_i^s.\mathsf{FT}[u]$
    - **if** $(esk_j \in \vec{s}_u)$**:**
      - $\pi_i^s.\mathsf{FT}[u] \leftarrow \emptyset$
- **for** $j \in \{1, ..., n_P\}$**:**
  - **for** $t \in \{1, ..., n_S\}$**:**
    - **if** $(\mathsf{PrevPart}(\pi_i^s, \pi_j^t))$**:**
      - **for** $u \in \{1, ..., |\pi_j^t.\mathsf{FT}|\}$**:**
        - $(\vec{s}_u, \mathtt{cf}_u, \mathtt{au}_j^{\mathtt{i}}, \mathtt{au}_j^{\mathtt{r}}) \leftarrow \pi_j^t.\mathsf{FT}[u]$
        - **if** $(esk_i \in \vec{s}_u) \wedge (\mathsf{comb} = 0)$**:**
          - $\pi_i^s.\mathsf{FT}[u] \leftarrow \emptyset$
        - **if** $(corr_i) \wedge (\mathsf{comb}) \wedge (esk_i \in \vec{s}_u)$**:**
          - $\pi_j^t.\mathsf{FT}[u] \leftarrow \emptyset$
      - **for** $\varsigma \in \{1, ..., n_T\}$**:**
        - **if** $(\nexists\ \mathtt{cf}_u \in \pi_j^t.\mathsf{FT} : \varsigma \geq \mathtt{cf}_u)$**:**
          - $\pi_j^t.fr_\varsigma \leftarrow 0.$
- **for** $\varsigma \in \{1, ..., n_T\}$**:**
  - **if** $(\nexists\ \mathtt{cf}_u \in \pi_j^t.\mathsf{FT} : \varsigma \geq \mathtt{cf}_u)$**:**
    - $\pi_j^t.fr_\varsigma \leftarrow 0.$
- **return** $\pi_i^s.rand$

Algorithm 12: PRP-SEEC

```
1 Function GenKey(1^λ):
2     return PRP.gen()
3 Function GenRand(sk, r):
4     return PRP.enc(sk, r), sk
```

independent value results in a random and independent value. Thus the adversary receives identically distributed values independent of the challenge-bit and the adversarial advantage is 0.

In the second case we use game-hopping to show the theorem. Let ***Game 0*** refer to the original SEEC-game with the provision that $\mathcal{A}$ never corrupts the static key.

In ***Game 1*** we abort if the ephemeral randomness ever collides. Given that the ephemeral randomness consists of truly random (though possibly known to $\mathcal{A}$) and independent bitstrings of length λ, the probability of a collision in n queries is $\leq \frac{n^2}{2^\lambda}$. Thus we find that:

$$\Pr[\text{break}_0] \leq \Pr[\text{break}_1] + \frac{n^2}{2^\lambda}$$

In ***Game 2*** we return truly random bitstrings instead of encryptions of the ephemeral randomness. To show that this replacement is sound we initialize an IND-CPA-challenger for PRP and use its encryption-oracle whenever we have to encrypt a static value and its challenge-oracle with the ephemeral entropy and a truly random value when answering $\mathcal{A}$'s challenge-query. As the static key is random and independent and since the queries don't repeat by *Game 1*, this substitution is valid. If the IND-CPA-challengers challenge bit is 0 then it returns an encryption of the ephemeral entropy and we are in *Game 1*. Otherwise it returns the result of applying a permutation to a random and independent value, which is in turn a random and independent value and we are in *Game 2* and find:

$$\Pr[\text{break}_1] \leq \Pr[\text{break}_2] + \mathsf{Adv}^{\mathsf{IND\text{-}CPA}}_{PRP,\mathcal{A}'}(1^\lambda)$$

In ***Game 3*** we abort if the output-randomness ever collides. Given that by *Game 2* the output-randomness consists of truly random and independent bitstrings of length λ, the probability of a collision in n queries is $\leq \frac{n^2}{2^\lambda}$. Thus we find that:

$$\Pr[\text{break}_2] \leq \Pr[\text{break}_3] + \frac{n^2}{2^\lambda}$$

At this point the SEEC-challenger always returns random values that don't repeat and thus there is no more information-flow from the challenge-bit b to $\mathcal{A}$ and we find:

$$\Pr[\text{break}_3] = 0$$

By summarizing all losses we find the combined loss stated in the theorem. □

B.3. Detailed Patterns

As outlined in Section 5.2 the main-difference between classical Noise and PQNoise lies in the `ekem` and `skem` operations, that we describe there. In addition to those PQNoise inherits the ability to send ephemeral (`e`) and static (`s`) public keys, the key-generation and the session initialization.

The key-generation of PQNoise (Algorithm 13) consists of up to three operations that a party may or may not perform, depending on the setting:

- The generation of a SEEC-key, if the use of SEEC is desired.
- The generation of a long-term static key for the party's KEM; if IKEM and RKEM are different, all parties that want to participate in both roles have to generate their static keys here, but we will for simplicity assume in the following that this is not the case, and each party just generates one key used for both purposes. If a party will never authenticate itself, it may skip this step.
- The authenticated distribution of the static public key, if one is generated and assumed to be known to the peer in the protocol (`*K` and `K*`-patterns). We denote this by a call to a function Publish, with the understanding that this is not so much an algorithm, but merely notation to indicate actions that have to happen out of band and are assumed to have been successful, by the time the parties start interacting with each other.

The session-initialization (Algorithm 14) of PQNoise essentially consists of the initialization of the hash-chains ck and h with values derived from the name of the pattern in question.

Calls to Send and Recv (Algorithm 15) will generally behave differently, depending on the protocol stage. Send will always start by setting up a payload-buffer pl and a send-buffer buf both initially empty. pl will be used to store any temporary values that may need to get encrypted, whereas buf

Algorithm 13: Key-Generation; Publish is skipped if the key is not known to the peer ahead of time.

```
1 Function KGen():
2     seec_sk ← SEEC.gen_key()
3     pk, sk ← XKEM.gen()
4     Publish(pk)
```

Algorithm 14: initialization

```
1 Function Init():
2     h ← H("pq**_label")
3     ck ← HashObject.gen("pq**_label")
```

will contain everything that has been processed completely and will in the end be sent as is over the network. After operations such as sending public keys (`e` and `s`, see below) and KEM-ciphertexts (`ekem` and `skem`, see Section 5.2) have been processed, the message m (which may be empty) will be appended to the payload. Then pl will be encrypted with the stage key if one exists, the resulting ciphertext added to buf and the hash-object h and the nonce n will be incremented. Otherwise pl will be added to buf and hashed into h as is. In either case buf will be what is finally sent over the wire. Recv largely mirrors that behavior, except that it receives buf and will extract one payload pl after another from it. If the first element of buf is encrypted, the receiver will decrypt it with the latest known AEAD-key, producing the first payload pl. Otherwise the plaintext will serve the role as initial pl. In the end the last element of the final payload pl will be considered the message and returned to the receiver.

Sending and receiving of public keys (Algorithm 16) consists of adding the keys in question to the current payload pl and of reading them from the current plaintext-buffer buf respectively. In the case of ephemeral public keys they are also generated as part of Send, optionally using SEEC.

With this we present all fundamental PQNoise-patterns in the Listings B.1–B.13. For the sake of simplicity we leave out passing and returning the state of the involved parties and just assume that they maintain it. For the same reason we also provide multiple definitions of all functions, per role and stage during the protocol-execution instead of just one that starts with a case-distinction that selects the appropriate version. This allows us to present the

Algorithm 15: Basic Send- and Receive-operations.

```
1  Function Send(m):
2      pl ← String.new()
3      buf ← String.new()
4      ...
5      if k_i ≠ ⊥:
6          c ← AEAD.enc(k_i, pl, h, n)
7          n.increment()
8          buf.add(c)
9          h ← H(h, c)
10     else:
11         buf.add(pl)
12         h ← H(h, pl)
13     return buf
14 Function Recv:
15     if k_i ≠ ⊥:
16         c ← buf.parse_next()
17         pl ← AEAD.dec(k_i, c, h, n)
18         h ← H(h, c)
19         n.increment()
20     else:
21         pl ← buf.parse_next()
22         h ← H(h, pl)
23     ...
24     m ← pl.parse_next()
25     return m
```

Algorithm 16: Sending and Receiving public keys.

Function Send:
    ...
    if $XKEM = EKEM$:
        $r \leftarrow SEEC.\mathsf{GenRand}(seec\_sk)$
        $pk_e, sk_e := XKEM.\mathsf{gen}(r)$
    $pl.add(pk_e)$
    ...
Function Recv:
    ...
    $pk_e \leftarrow pl.\mathsf{parse\_next}()$
    ...

different versions that are used during a key-exchange in the order in which honest parties would execute them.

We remark that these patterns are auto-generated with a tool that we wrote and as a result of that slightly more verbose than strictly-speaking necessary. The advantage of that approach is however that it ensures consistency between the patterns and allowed us to write a type-checker that confirmed that they do not contain obvious type-errors, such as ciphertexts being generated using one algorithm and then later decrypted by another.

Listing B.1: PQN

```
def keygen_init():
    seec_sk = SEEC.gen_key()

def keygen_resp():
    seec_sk = SEEC.gen_key()
    pk_resp, sk_resp = RKem.gen()
    Publish(pk_resp)

def initialize_init():
    h = Hash("pqN_label")
    ck = HashObject.gen("pqN_label")

def send_init_1(m):
    payload = ""
    buffer = ""
    r = SEEC.gen_rand(seec_sk)
    ct_resp, k_resp = RKem.enc(pk_resp, r)
    payload.add(ct_resp)
```

```
    h = Hash(h, payload)
    buffer.add(payload)
    payload.flush()
    prekey_init, prekey_resp = HashObject.finalize(ck, k_resp)
    key_init = AEAD.gen(prekey_init)
    key_resp = AEAD.gen(prekey_resp)
    nonce = AEAD.Nonce.new()
    payload.add(m)
    c = AEAD.enc(key_init, payload, h, nonce)
    nonce.increment()
    buffer.add(c)
    h = Hash(h, c)
    return buffer

def initialize_resp():
    h = Hash("pqN_label")
    ck = HashObject.gen("pqN_label")

def receive_resp_1(buffer):
    payload = buffer.parse_next()
    h = Hash(h, payload)
    ct_resp = payload.parse_next()
    k_resp = RKem.dec(sk_resp, ct_resp)
    prekey_init, prekey_resp = HashObject.finalize(ck, k_resp)
    key_init = AEAD.gen(prekey_init)
    key_resp = AEAD.gen(prekey_resp)
    nonce = AEAD.Nonce.new()
    c = buffer.parse_next()
    payload = AEAD.dec(key_init, c, h, nonce)
    h = Hash(h, c)
    nonce.increment()
    m = payload.parse_next()
    return m
```

Listing B.2: PQNN

```
def keygen_init():
    seec_sk = SEEC.gen_key()

def keygen_resp():
    seec_sk = SEEC.gen_key()

def initialize_init():
    h = Hash("pqNN_label")
    ck = HashObject.gen("pqNN_label")

def send_init_1(m):
    payload = ""
    buffer = ""
    r = SEEC.gen_rand(seec_sk)
```

```
    pk_e, sk_e = EKem.gen(r)
    payload.add(pk_e)
    payload.add(m)
    buffer.add(payload)
    h = Hash(h, payload)
    return buffer

def initialize_resp():
    h = Hash("pqNN_label")
    ck = HashObject.gen("pqNN_label")

def receive_resp_1(buffer):
    payload = buffer.parse_next()
    h = Hash(h, payload)
    pk_e = payload.parse_next()
    m = payload.parse_next()
    return m

def send_resp_1(m):
    payload = ""
    buffer = ""
    r = SEEC.gen_rand(seec_sk)
    ct_e, k_e = EKem.enc(pk_e, r)
    payload.add(ct_e)
    h = Hash(h, payload)
    buffer.add(payload)
    payload.flush()
    prekey_init, prekey_resp = HashObject.finalize(ck, k_e)
    key_init = AEAD.gen(prekey_init)
    key_resp = AEAD.gen(prekey_resp)
    nonce = AEAD.Nonce.new()
    payload.add(m)
    c = AEAD.enc(key_resp, payload, h, nonce)
    nonce.increment()
    buffer.add(c)
    h = Hash(h, c)
    return buffer

def receive_init_1(buffer):
    payload = buffer.parse_next()
    h = Hash(h, payload)
    ct_e = payload.parse_next()
    k_e = EKem.dec(sk_e, ct_e)
    prekey_init, prekey_resp = HashObject.finalize(ck, k_e)
    key_init = AEAD.gen(prekey_init)
    key_resp = AEAD.gen(prekey_resp)
    nonce = AEAD.Nonce.new()
    c = buffer.parse_next()
    payload = AEAD.dec(key_resp, c, h, nonce)
    h = Hash(h, c)
```

```
    nonce.increment()
    m = payload.parse_next()
    return m
```

Listing B.3: PQNK

```
def keygen_init():
    seec_sk = SEEC.gen_key()

def keygen_resp():
    seec_sk = SEEC.gen_key()
    pk_resp, sk_resp = RKem.gen()
    Publish(pk_resp)

def initialize_init():
    h = Hash("pqNK_label")
    ck = HashObject.gen("pqNK_label")

def send_init_1(m):
    payload = ""
    buffer = ""
    r = SEEC.gen_rand(seec_sk)
    ct_resp, k_resp = RKem.enc(pk_resp, r)
    payload.add(ct_resp)
    h = Hash(h, payload)
    buffer.add(payload)
    payload.flush()
    prekey = ck.input(k_resp)
    key_0 = AEAD.gen(prekey)
    nonce = AEAD.Nonce.new()
    r = SEEC.gen_rand(seec_sk)
    pk_e, sk_e = EKem.gen(r)
    payload.add(pk_e)
    payload.add(m)
    c = AEAD.enc(key_0, payload, h, nonce)
    nonce.increment()
    buffer.add(c)
    h = Hash(h, c)
    return buffer

def initialize_resp():
    h = Hash("pqNK_label")
    ck = HashObject.gen("pqNK_label")

def receive_resp_1(buffer):
    payload = buffer.parse_next()
    h = Hash(h, payload)
    ct_resp = payload.parse_next()
    k_resp = RKem.dec(sk_resp, ct_resp)
    prekey = ck.input(k_resp)
```

```
    key_0 = AEAD.gen(prekey)
    nonce = AEAD.Nonce.new()
    c = buffer.parse_next()
    payload = AEAD.dec(key_0, c, h, nonce)
    h = Hash(h, c)
    nonce.increment()
    pk_e = payload.parse_next()
    m = payload.parse_next()
    return m

def send_resp_1(m):
    payload = ""
    buffer = ""
    r = SEEC.gen_rand(seec_sk)
    ct_e, k_e = EKem.enc(pk_e, r)
    payload.add(ct_e)
    h = Hash(h, payload)
    buffer.add(payload)
    payload.flush()
    prekey_init, prekey_resp = HashObject.finalize(ck, k_e)
    key_init = AEAD.gen(prekey_init)
    key_resp = AEAD.gen(prekey_resp)
    nonce = AEAD.Nonce.new()
    payload.add(m)
    c = AEAD.enc(key_resp, payload, h, nonce)
    nonce.increment()
    buffer.add(c)
    h = Hash(h, c)
    return buffer

def receive_init_1(buffer):
    c = buffer.parse_next()
    payload = AEAD.dec(key_0, c, h, nonce)
    h = Hash(h, c)
    nonce.increment()
    ct_e = payload.parse_next()
    k_e = EKem.dec(sk_e, ct_e)
    prekey_init, prekey_resp = HashObject.finalize(ck, k_e)
    key_init = AEAD.gen(prekey_init)
    key_resp = AEAD.gen(prekey_resp)
    nonce = AEAD.Nonce.new()
    c = buffer.parse_next()
    payload = AEAD.dec(key_resp, c, h, nonce)
    h = Hash(h, c)
    nonce.increment()
    m = payload.parse_next()
    return m
```

Listing B.4: PQNX

```
def keygen_init():
    seec_sk = SEEC.gen_key()

def keygen_resp():
    seec_sk = SEEC.gen_key()
    pk_resp, sk_resp = RKem.gen()

def initialize_init():
    h = Hash("pqNX_label")
    ck = HashObject.gen("pqNX_label")

def send_init_1(m):
    payload = ""
    buffer = ""
    r = SEEC.gen_rand(seec_sk)
    pk_e, sk_e = EKem.gen(r)
    payload.add(pk_e)
    payload.add(m)
    buffer.add(payload)
    h = Hash(h, payload)
    return buffer

def initialize_resp():
    h = Hash("pqNX_label")
    ck = HashObject.gen("pqNX_label")

def receive_resp_1(buffer):
    payload = buffer.parse_next()
    h = Hash(h, payload)
    pk_e = payload.parse_next()
    m = payload.parse_next()
    return m

def send_resp_1(m):
    payload = ""
    buffer = ""
    r = SEEC.gen_rand(seec_sk)
    ct_e, k_e = EKem.enc(pk_e, r)
    payload.add(ct_e)
    h = Hash(h, payload)
    buffer.add(payload)
    payload.flush()
    prekey = ck.input(k_e)
    key_0 = AEAD.gen(prekey)
    nonce = AEAD.Nonce.new()
    payload.add(pk_resp)
    payload.add(m)
    c = AEAD.enc(key_0, payload, h, nonce)
    nonce.increment()
```

```
    buffer.add(c)
    h = Hash(h, c)
    return buffer

def receive_init_1(buffer):
    payload = buffer.parse_next()
    h = Hash(h, payload)
    ct_e = payload.parse_next()
    k_e = EKem.dec(sk_e, ct_e)
    prekey = ck.input(k_e)
    key_0 = AEAD.gen(prekey)
    nonce = AEAD.Nonce.new()
    c = buffer.parse_next()
    payload = AEAD.dec(key_0, c, h, nonce)
    h = Hash(h, c)
    nonce.increment()
    pk_resp = payload.parse_next()
    m = payload.parse_next()
    return m

def send_init_2(m):
    payload = ""
    buffer = ""
    r = SEEC.gen_rand(seec_sk)
    ct_resp, k_resp = RKem.enc(pk_resp, r)
    payload.add(ct_resp)
    c = AEAD.enc(key_0, payload, h, nonce)
    nonce.increment()
    h = Hash(h, c)
    buffer.add(c)
    payload.flush()
    prekey_init, prekey_resp = HashObject.finalize(ck, k_resp)
    key_init = AEAD.gen(prekey_init)
    key_resp = AEAD.gen(prekey_resp)
    nonce = AEAD.Nonce.new()
    payload.add(m)
    c = AEAD.enc(key_init, payload, h, nonce)
    nonce.increment()
    buffer.add(c)
    h = Hash(h, c)
    return buffer

def receive_resp_2(buffer):
    c = buffer.parse_next()
    payload = AEAD.dec(key_0, c, h, nonce)
    h = Hash(h, c)
    nonce.increment()
    ct_resp = payload.parse_next()
    k_resp = RKem.dec(sk_resp, ct_resp)
    prekey_init, prekey_resp = HashObject.finalize(ck, k_resp)
```

```
    key_init = AEAD.gen(prekey_init)
    key_resp = AEAD.gen(prekey_resp)
    nonce = AEAD.Nonce.new()
    c = buffer.parse_next()
    payload = AEAD.dec(key_init, c, h, nonce)
    h = Hash(h, c)
    nonce.increment()
    m = payload.parse_next()
    return m
```

Listing B.5: PQKN

```
def keygen_init():
    seec_sk = SEEC.gen_key()
    pk_init, sk_init = IKem.gen()
    Publish(pk_init)

def keygen_resp():
    seec_sk = SEEC.gen_key()

def initialize_init():
    h = Hash("pqKN_label")
    ck = HashObject.gen("pqKN_label")

def send_init_1(m):
    payload = ""
    buffer = ""
    r = SEEC.gen_rand(seec_sk)
    pk_e, sk_e = EKem.gen(r)
    payload.add(pk_e)
    payload.add(m)
    buffer.add(payload)
    h = Hash(h, payload)
    return buffer

def initialize_resp():
    h = Hash("pqKN_label")
    ck = HashObject.gen("pqKN_label")

def receive_resp_1(buffer):
    payload = buffer.parse_next()
    h = Hash(h, payload)
    pk_e = payload.parse_next()
    m = payload.parse_next()
    return m

def send_resp_1(m):
    payload = ""
    buffer = ""
    r = SEEC.gen_rand(seec_sk)
```

```
    ct_e, k_e = EKem.enc(pk_e, r)
    payload.add(ct_e)
    h = Hash(h, payload)
    buffer.add(payload)
    payload.flush()
    prekey = ck.input(k_e)
    key_0 = AEAD.gen(prekey)
    nonce = AEAD.Nonce.new()
    r = SEEC.gen_rand(seec_sk)
    ct_init, k_init = IKem.enc(pk_init, r)
    payload.add(ct_init)
    c = AEAD.enc(key_0, payload, h, nonce)
    nonce.increment()
    h = Hash(h, c)
    buffer.add(c)
    payload.flush()
    prekey_init, prekey_resp = HashObject.finalize(ck, k_init)
    key_init = AEAD.gen(prekey_init)
    key_resp = AEAD.gen(prekey_resp)
    nonce = AEAD.Nonce.new()
    payload.add(m)
    c = AEAD.enc(key_resp, payload, h, nonce)
    nonce.increment()
    buffer.add(c)
    h = Hash(h, c)
    return buffer

def receive_init_1(buffer):
    payload = buffer.parse_next()
    h = Hash(h, payload)
    ct_e = payload.parse_next()
    k_e = EKem.dec(sk_e, ct_e)
    prekey = ck.input(k_e)
    key_0 = AEAD.gen(prekey)
    nonce = AEAD.Nonce.new()
    c = buffer.parse_next()
    payload = AEAD.dec(key_0, c, h, nonce)
    h = Hash(h, c)
    nonce.increment()
    ct_init = payload.parse_next()
    k_init = IKem.dec(sk_init, ct_init)
    prekey_init, prekey_resp = HashObject.finalize(ck, k_init)
    key_init = AEAD.gen(prekey_init)
    key_resp = AEAD.gen(prekey_resp)
    nonce = AEAD.Nonce.new()
    c = buffer.parse_next()
    payload = AEAD.dec(key_resp, c, h, nonce)
    h = Hash(h, c)
    nonce.increment()
    m = payload.parse_next()
```

```
89      return m
```

Listing B.6: PQKK

```
1  def keygen_init():
2      seec_sk = SEEC.gen_key()
3      pk_init, sk_init = IKem.gen()
4      Publish(pk_init)
5
6  def keygen_resp():
7      seec_sk = SEEC.gen_key()
8      pk_resp, sk_resp = RKem.gen()
9      Publish(pk_resp)
10
11 def initialize_init():
12     h = Hash("pqKK_label")
13     ck = HashObject.gen("pqKK_label")
14
15 def send_init_1(m):
16     payload = ""
17     buffer = ""
18     r = SEEC.gen_rand(seec_sk)
19     ct_resp, k_resp = RKem.enc(pk_resp, r)
20     payload.add(ct_resp)
21     h = Hash(h, payload)
22     buffer.add(payload)
23     payload.flush()
24     prekey = ck.input(k_resp)
25     key_0 = AEAD.gen(prekey)
26     nonce = AEAD.Nonce.new()
27     r = SEEC.gen_rand(seec_sk)
28     pk_e, sk_e = EKem.gen(r)
29     payload.add(pk_e)
30     payload.add(m)
31     c = AEAD.enc(key_0, payload, h, nonce)
32     nonce.increment()
33     buffer.add(c)
34     h = Hash(h, c)
35     return buffer
36
37 def initialize_resp():
38     h = Hash("pqKK_label")
39     ck = HashObject.gen("pqKK_label")
40
41 def receive_resp_1(buffer):
42     payload = buffer.parse_next()
43     h = Hash(h, payload)
44     ct_resp = payload.parse_next()
45     k_resp = RKem.dec(sk_resp, ct_resp)
46     prekey = ck.input(k_resp)
```

```
47      key_0 = AEAD.gen(prekey)
48      nonce = AEAD.Nonce.new()
49      c = buffer.parse_next()
50      payload = AEAD.dec(key_0, c, h, nonce)
51      h = Hash(h, c)
52      nonce.increment()
53      pk_e = payload.parse_next()
54      m = payload.parse_next()
55      return m
56
57  def send_resp_1(m):
58      payload = ""
59      buffer = ""
60      r = SEEC.gen_rand(seec_sk)
61      ct_e, k_e = EKem.enc(pk_e, r)
62      payload.add(ct_e)
63      h = Hash(h, payload)
64      buffer.add(payload)
65      payload.flush()
66      prekey = ck.input(k_e)
67      key_1 = AEAD.gen(prekey)
68      nonce = AEAD.Nonce.new()
69      r = SEEC.gen_rand(seec_sk)
70      ct_init, k_init = IKem.enc(pk_init, r)
71      payload.add(ct_init)
72      c = AEAD.enc(key_1, payload, h, nonce)
73      nonce.increment()
74      h = Hash(h, c)
75      buffer.add(c)
76      payload.flush()
77      prekey_init, prekey_resp = HashObject.finalize(ck, k_init)
78      key_init = AEAD.gen(prekey_init)
79      key_resp = AEAD.gen(prekey_resp)
80      nonce = AEAD.Nonce.new()
81      payload.add(m)
82      c = AEAD.enc(key_resp, payload, h, nonce)
83      nonce.increment()
84      buffer.add(c)
85      h = Hash(h, c)
86      return buffer
87
88  def receive_init_1(buffer):
89      c = buffer.parse_next()
90      payload = AEAD.dec(key_0, c, h, nonce)
91      h = Hash(h, c)
92      nonce.increment()
93      ct_e = payload.parse_next()
94      k_e = EKem.dec(sk_e, ct_e)
95      prekey = ck.input(k_e)
96      key_1 = AEAD.gen(prekey)
```

```
    nonce = AEAD.Nonce.new()
    c = buffer.parse_next()
    payload = AEAD.dec(key_0, c, h, nonce)
    h = Hash(h, c)
    nonce.increment()
    ct_init = payload.parse_next()
    k_init = IKem.dec(sk_init, ct_init)
    prekey_init, prekey_resp = HashObject.finalize(ck, k_init)
    key_init = AEAD.gen(prekey_init)
    key_resp = AEAD.gen(prekey_resp)
    nonce = AEAD.Nonce.new()
    c = buffer.parse_next()
    payload = AEAD.dec(key_resp, c, h, nonce)
    h = Hash(h, c)
    nonce.increment()
    m = payload.parse_next()
    return m
```

Listing B.7: PQKX

```
def keygen_init():
    seec_sk = SEEC.gen_key()
    pk_init, sk_init = IKem.gen()
    Publish(pk_init)

def keygen_resp():
    seec_sk = SEEC.gen_key()
    pk_resp, sk_resp = RKem.gen()

def initialize_init():
    h = Hash("pqKX_label")
    ck = HashObject.gen("pqKX_label")

def send_init_1(m):
    payload = ""
    buffer = ""
    r = SEEC.gen_rand(seec_sk)
    pk_e, sk_e = EKem.gen(r)
    payload.add(pk_e)
    payload.add(m)
    buffer.add(payload)
    h = Hash(h, payload)
    return buffer

def initialize_resp():
    h = Hash("pqKX_label")
    ck = HashObject.gen("pqKX_label")

def receive_resp_1(buffer):
    payload = buffer.parse_next()
```

```
31      h = Hash(h, payload)
32      pk_e = payload.parse_next()
33      m = payload.parse_next()
34      return m
35
36  def send_resp_1(m):
37      payload = ""
38      buffer = ""
39      r = SEEC.gen_rand(seec_sk)
40      ct_e, k_e = EKem.enc(pk_e, r)
41      payload.add(ct_e)
42      h = Hash(h, payload)
43      buffer.add(payload)
44      payload.flush()
45      prekey = ck.input(k_e)
46      key_0 = AEAD.gen(prekey)
47      nonce = AEAD.Nonce.new()
48      r = SEEC.gen_rand(seec_sk)
49      ct_init, k_init = IKem.enc(pk_init, r)
50      payload.add(ct_init)
51      c = AEAD.enc(key_0, payload, h, nonce)
52      nonce.increment()
53      h = Hash(h, c)
54      buffer.add(c)
55      payload.flush()
56      prekey = ck.input(k_init)
57      key_1 = AEAD.gen(prekey)
58      nonce = AEAD.Nonce.new()
59      payload.add(pk_resp)
60      payload.add(m)
61      c = AEAD.enc(key_1, payload, h, nonce)
62      nonce.increment()
63      buffer.add(c)
64      h = Hash(h, c)
65      return buffer
66
67  def receive_init_1(buffer):
68      payload = buffer.parse_next()
69      h = Hash(h, payload)
70      ct_e = payload.parse_next()
71      k_e = EKem.dec(sk_e, ct_e)
72      prekey = ck.input(k_e)
73      key_0 = AEAD.gen(prekey)
74      nonce = AEAD.Nonce.new()
75      c = buffer.parse_next()
76      payload = AEAD.dec(key_0, c, h, nonce)
77      h = Hash(h, c)
78      nonce.increment()
79      ct_init = payload.parse_next()
80      k_init = IKem.dec(sk_init, ct_init)
```

```
    prekey = ck.input(k_init)
    key_1 = AEAD.gen(prekey)
    nonce = AEAD.Nonce.new()
    c = buffer.parse_next()
    payload = AEAD.dec(key_0, c, h, nonce)
    h = Hash(h, c)
    nonce.increment()
    pk_resp = payload.parse_next()
    m = payload.parse_next()
    return m

def send_init_2(m):
    payload = ""
    buffer = ""
    r = SEEC.gen_rand(seec_sk)
    ct_resp, k_resp = RKem.enc(pk_resp, r)
    payload.add(ct_resp)
    c = AEAD.enc(key_1, payload, h, nonce)
    nonce.increment()
    h = Hash(h, c)
    buffer.add(c)
    payload.flush()
    prekey_init, prekey_resp = HashObject.finalize(ck, k_resp)
    key_init = AEAD.gen(prekey_init)
    key_resp = AEAD.gen(prekey_resp)
    nonce = AEAD.Nonce.new()
    payload.add(m)
    c = AEAD.enc(key_init, payload, h, nonce)
    nonce.increment()
    buffer.add(c)
    h = Hash(h, c)
    return buffer

def receive_resp_2(buffer):
    c = buffer.parse_next()
    payload = AEAD.dec(key_1, c, h, nonce)
    h = Hash(h, c)
    nonce.increment()
    ct_resp = payload.parse_next()
    k_resp = RKem.dec(sk_resp, ct_resp)
    prekey_init, prekey_resp = HashObject.finalize(ck, k_resp)
    key_init = AEAD.gen(prekey_init)
    key_resp = AEAD.gen(prekey_resp)
    nonce = AEAD.Nonce.new()
    c = buffer.parse_next()
    payload = AEAD.dec(key_init, c, h, nonce)
    h = Hash(h, c)
    nonce.increment()
    m = payload.parse_next()
    return m
```

Listing B.8: PQXN

```
def keygen_init():
    seec_sk = SEEC.gen_key()
    pk_init, sk_init = IKem.gen()

def keygen_resp():
    seec_sk = SEEC.gen_key()

def initialize_init():
    h = Hash("pqXN_label")
    ck = HashObject.gen("pqXN_label")

def send_init_1(m):
    payload = ""
    buffer = ""
    r = SEEC.gen_rand(seec_sk)
    pk_e, sk_e = EKem.gen(r)
    payload.add(pk_e)
    payload.add(m)
    buffer.add(payload)
    h = Hash(h, payload)
    return buffer

def initialize_resp():
    h = Hash("pqXN_label")
    ck = HashObject.gen("pqXN_label")

def receive_resp_1(buffer):
    payload = buffer.parse_next()
    h = Hash(h, payload)
    pk_e = payload.parse_next()
    m = payload.parse_next()
    return m

def send_resp_1(m):
    payload = ""
    buffer = ""
    r = SEEC.gen_rand(seec_sk)
    ct_e, k_e = EKem.enc(pk_e, r)
    payload.add(ct_e)
    h = Hash(h, payload)
    buffer.add(payload)
    payload.flush()
    prekey = ck.input(k_e)
    key_0 = AEAD.gen(prekey)
    nonce = AEAD.Nonce.new()
    payload.add(m)
```

```
    c = AEAD.enc(key_0, payload, h, nonce)
    nonce.increment()
    buffer.add(c)
    h = Hash(h, c)
    return buffer

def receive_init_1(buffer):
    payload = buffer.parse_next()
    h = Hash(h, payload)
    ct_e = payload.parse_next()
    k_e = EKem.dec(sk_e, ct_e)
    prekey = ck.input(k_e)
    key_0 = AEAD.gen(prekey)
    nonce = AEAD.Nonce.new()
    c = buffer.parse_next()
    payload = AEAD.dec(key_0, c, h, nonce)
    h = Hash(h, c)
    nonce.increment()
    m = payload.parse_next()
    return m

def send_init_2(m):
    payload = ""
    buffer = ""
    payload.add(pk_init)
    payload.add(m)
    c = AEAD.enc(key_0, payload, h, nonce)
    nonce.increment()
    buffer.add(c)
    h = Hash(h, c)
    return buffer

def receive_resp_2(buffer):
    c = buffer.parse_next()
    payload = AEAD.dec(key_0, c, h, nonce)
    h = Hash(h, c)
    nonce.increment()
    pk_init = payload.parse_next()
    m = payload.parse_next()
    return m

def send_resp_2(m):
    payload = ""
    buffer = ""
    r = SEEC.gen_rand(seec_sk)
    ct_init, k_init = IKem.enc(pk_init, r)
    payload.add(ct_init)
    c = AEAD.enc(key_0, payload, h, nonce)
    nonce.increment()
    h = Hash(h, c)
```

```
    buffer.add(c)
    payload.flush()
    prekey_init, prekey_resp = HashObject.finalize(ck, k_init)
    key_init = AEAD.gen(prekey_init)
    key_resp = AEAD.gen(prekey_resp)
    nonce = AEAD.Nonce.new()
    payload.add(m)
    c = AEAD.enc(key_resp, payload, h, nonce)
    nonce.increment()
    buffer.add(c)
    h = Hash(h, c)
    return buffer

def receive_init_2(buffer):
    c = buffer.parse_next()
    payload = AEAD.dec(key_0, c, h, nonce)
    h = Hash(h, c)
    nonce.increment()
    ct_init = payload.parse_next()
    k_init = IKem.dec(sk_init, ct_init)
    prekey_init, prekey_resp = HashObject.finalize(ck, k_init)
    key_init = AEAD.gen(prekey_init)
    key_resp = AEAD.gen(prekey_resp)
    nonce = AEAD.Nonce.new()
    c = buffer.parse_next()
    payload = AEAD.dec(key_resp, c, h, nonce)
    h = Hash(h, c)
    nonce.increment()
    m = payload.parse_next()
    return m
```

Listing B.9: PQXK

```
def keygen_init():
    seec_sk = SEEC.gen_key()
    pk_init, sk_init = IKem.gen()

def keygen_resp():
    seec_sk = SEEC.gen_key()
    pk_resp, sk_resp = RKem.gen()
    Publish(pk_resp)

def initialize_init():
    h = Hash("pqXK_label")
    ck = HashObject.gen("pqXK_label")

def send_init_1(m):
    payload = ""
    buffer = ""
    r = SEEC.gen_rand(seec_sk)
```

```
    ct_resp, k_resp = RKem.enc(pk_resp, r)
    payload.add(ct_resp)
    h = Hash(h, payload)
    buffer.add(payload)
    payload.flush()
    prekey = ck.input(k_resp)
    key_0 = AEAD.gen(prekey)
    nonce = AEAD.Nonce.new()
    r = SEEC.gen_rand(seec_sk)
    pk_e, sk_e = EKem.gen(r)
    payload.add(pk_e)
    payload.add(m)
    c = AEAD.enc(key_0, payload, h, nonce)
    nonce.increment()
    buffer.add(c)
    h = Hash(h, c)
    return buffer

def initialize_resp():
    h = Hash("pqXK_label")
    ck = HashObject.gen("pqXK_label")

def receive_resp_1(buffer):
    payload = buffer.parse_next()
    h = Hash(h, payload)
    ct_resp = payload.parse_next()
    k_resp = RKem.dec(sk_resp, ct_resp)
    prekey = ck.input(k_resp)
    key_0 = AEAD.gen(prekey)
    nonce = AEAD.Nonce.new()
    c = buffer.parse_next()
    payload = AEAD.dec(key_0, c, h, nonce)
    h = Hash(h, c)
    nonce.increment()
    pk_e = payload.parse_next()
    m = payload.parse_next()
    return m

def send_resp_1(m):
    payload = ""
    buffer = ""
    r = SEEC.gen_rand(seec_sk)
    ct_e, k_e = EKem.enc(pk_e, r)
    payload.add(ct_e)
    h = Hash(h, payload)
    buffer.add(payload)
    payload.flush()
    prekey = ck.input(k_e)
    key_1 = AEAD.gen(prekey)
    nonce = AEAD.Nonce.new()
```

```
    payload.add(m)
    c = AEAD.enc(key_1, payload, h, nonce)
    nonce.increment()
    buffer.add(c)
    h = Hash(h, c)
    return buffer

def receive_init_1(buffer):
    c = buffer.parse_next()
    payload = AEAD.dec(key_0, c, h, nonce)
    h = Hash(h, c)
    nonce.increment()
    ct_e = payload.parse_next()
    k_e = EKem.dec(sk_e, ct_e)
    prekey = ck.input(k_e)
    key_1 = AEAD.gen(prekey)
    nonce = AEAD.Nonce.new()
    c = buffer.parse_next()
    payload = AEAD.dec(key_0, c, h, nonce)
    h = Hash(h, c)
    nonce.increment()
    m = payload.parse_next()
    return m

def send_init_2(m):
    payload = ""
    buffer = ""
    payload.add(pk_init)
    payload.add(m)
    c = AEAD.enc(key_1, payload, h, nonce)
    nonce.increment()
    buffer.add(c)
    h = Hash(h, c)
    return buffer

def receive_resp_2(buffer):
    c = buffer.parse_next()
    payload = AEAD.dec(key_1, c, h, nonce)
    h = Hash(h, c)
    nonce.increment()
    pk_init = payload.parse_next()
    m = payload.parse_next()
    return m

def send_resp_2(m):
    payload = ""
    buffer = ""
    r = SEEC.gen_rand(seec_sk)
    ct_init, k_init = IKem.enc(pk_init, r)
    payload.add(ct_init)
```

```
    c = AEAD.enc(key_1, payload, h, nonce)
    nonce.increment()
    h = Hash(h, c)
    buffer.add(c)
    payload.flush()
    prekey_init, prekey_resp = HashObject.finalize(ck, k_init)
    key_init = AEAD.gen(prekey_init)
    key_resp = AEAD.gen(prekey_resp)
    nonce = AEAD.Nonce.new()
    payload.add(m)
    c = AEAD.enc(key_resp, payload, h, nonce)
    nonce.increment()
    buffer.add(c)
    h = Hash(h, c)
    return buffer

def receive_init_2(buffer):
    c = buffer.parse_next()
    payload = AEAD.dec(key_1, c, h, nonce)
    h = Hash(h, c)
    nonce.increment()
    ct_init = payload.parse_next()
    k_init = IKem.dec(sk_init, ct_init)
    prekey_init, prekey_resp = HashObject.finalize(ck, k_init)
    key_init = AEAD.gen(prekey_init)
    key_resp = AEAD.gen(prekey_resp)
    nonce = AEAD.Nonce.new()
    c = buffer.parse_next()
    payload = AEAD.dec(key_resp, c, h, nonce)
    h = Hash(h, c)
    nonce.increment()
    m = payload.parse_next()
    return m
```

Listing B.10: PQXX

```
def keygen_init():
    seec_sk = SEEC.gen_key()
    pk_init, sk_init = IKem.gen()

def keygen_resp():
    seec_sk = SEEC.gen_key()
    pk_resp, sk_resp = RKem.gen()

def initialize_init():
    h = Hash("pqXX_label")
    ck = HashObject.gen("pqXX_label")

def send_init_1(m):
    payload = ""
```

```
    buffer = ""
    r = SEEC.gen_rand(seec_sk)
    pk_e, sk_e = EKem.gen(r)
    payload.add(pk_e)
    payload.add(m)
    buffer.add(payload)
    h = Hash(h, payload)
    return buffer

def initialize_resp():
    h = Hash("pqXX_label")
    ck = HashObject.gen("pqXX_label")

def receive_resp_1(buffer):
    payload = buffer.parse_next()
    h = Hash(h, payload)
    pk_e = payload.parse_next()
    m = payload.parse_next()
    return m

def send_resp_1(m):
    payload = ""
    buffer = ""
    r = SEEC.gen_rand(seec_sk)
    ct_e, k_e = EKem.enc(pk_e, r)
    payload.add(ct_e)
    h = Hash(h, payload)
    buffer.add(payload)
    payload.flush()
    prekey = ck.input(k_e)
    key_0 = AEAD.gen(prekey)
    nonce = AEAD.Nonce.new()
    payload.add(pk_resp)
    payload.add(m)
    c = AEAD.enc(key_0, payload, h, nonce)
    nonce.increment()
    buffer.add(c)
    h = Hash(h, c)
    return buffer

def receive_init_1(buffer):
    payload = buffer.parse_next()
    h = Hash(h, payload)
    ct_e = payload.parse_next()
    k_e = EKem.dec(sk_e, ct_e)
    prekey = ck.input(k_e)
    key_0 = AEAD.gen(prekey)
    nonce = AEAD.Nonce.new()
    c = buffer.parse_next()
    payload = AEAD.dec(key_0, c, h, nonce)
```

```
    h = Hash(h, c)
    nonce.increment()
    pk_resp = payload.parse_next()
    m = payload.parse_next()
    return m

def send_init_2(m):
    payload = ""
    buffer = ""
    r = SEEC.gen_rand(seec_sk)
    ct_resp, k_resp = RKem.enc(pk_resp, r)
    payload.add(ct_resp)
    c = AEAD.enc(key_0, payload, h, nonce)
    nonce.increment()
    h = Hash(h, c)
    buffer.add(c)
    payload.flush()
    prekey = ck.input(k_resp)
    key_1 = AEAD.gen(prekey)
    nonce = AEAD.Nonce.new()
    payload.add(pk_init)
    payload.add(m)
    c = AEAD.enc(key_1, payload, h, nonce)
    nonce.increment()
    buffer.add(c)
    h = Hash(h, c)
    return buffer

def receive_resp_2(buffer):
    c = buffer.parse_next()
    payload = AEAD.dec(key_0, c, h, nonce)
    h = Hash(h, c)
    nonce.increment()
    ct_resp = payload.parse_next()
    k_resp = RKem.dec(sk_resp, ct_resp)
    prekey = ck.input(k_resp)
    key_1 = AEAD.gen(prekey)
    nonce = AEAD.Nonce.new()
    c = buffer.parse_next()
    payload = AEAD.dec(key_0, c, h, nonce)
    h = Hash(h, c)
    nonce.increment()
    pk_init = payload.parse_next()
    m = payload.parse_next()
    return m

def send_resp_2(m):
    payload = ""
    buffer = ""
    r = SEEC.gen_rand(seec_sk)
```

```
    ct_init, k_init = IKem.enc(pk_init, r)
    payload.add(ct_init)
    c = AEAD.enc(key_1, payload, h, nonce)
    nonce.increment()
    h = Hash(h, c)
    buffer.add(c)
    payload.flush()
    prekey_init, prekey_resp = HashObject.finalize(ck, k_init)
    key_init = AEAD.gen(prekey_init)
    key_resp = AEAD.gen(prekey_resp)
    nonce = AEAD.Nonce.new()
    payload.add(m)
    c = AEAD.enc(key_resp, payload, h, nonce)
    nonce.increment()
    buffer.add(c)
    h = Hash(h, c)
    return buffer

def receive_init_2(buffer):
    c = buffer.parse_next()
    payload = AEAD.dec(key_1, c, h, nonce)
    h = Hash(h, c)
    nonce.increment()
    ct_init = payload.parse_next()
    k_init = IKem.dec(sk_init, ct_init)
    prekey_init, prekey_resp = HashObject.finalize(ck, k_init)
    key_init = AEAD.gen(prekey_init)
    key_resp = AEAD.gen(prekey_resp)
    nonce = AEAD.Nonce.new()
    c = buffer.parse_next()
    payload = AEAD.dec(key_resp, c, h, nonce)
    h = Hash(h, c)
    nonce.increment()
    m = payload.parse_next()
    return m
```

Listing B.11: PQIN

```
def keygen_init():
    seec_sk = SEEC.gen_key()
    pk_init, sk_init = IKem.gen()

def keygen_resp():
    seec_sk = SEEC.gen_key()

def initialize_init():
    h = Hash("pqIN_label")
    ck = HashObject.gen("pqIN_label")

def send_init_1(m):
```

```
    payload = ""
    buffer = ""
    r = SEEC.gen_rand(seec_sk)
    pk_e, sk_e = EKem.gen(r)
    payload.add(pk_e)
    payload.add(pk_init)
    payload.add(m)
    buffer.add(payload)
    h = Hash(h, payload)
    return buffer

def initialize_resp():
    h = Hash("pqIN_label")
    ck = HashObject.gen("pqIN_label")

def receive_resp_1(buffer):
    payload = buffer.parse_next()
    h = Hash(h, payload)
    pk_e = payload.parse_next()
    pk_init = payload.parse_next()
    m = payload.parse_next()
    return m

def send_resp_1(m):
    payload = ""
    buffer = ""
    r = SEEC.gen_rand(seec_sk)
    ct_e, k_e = EKem.enc(pk_e, r)
    payload.add(ct_e)
    h = Hash(h, payload)
    buffer.add(payload)
    payload.flush()
    prekey = ck.input(k_e)
    key_0 = AEAD.gen(prekey)
    nonce = AEAD.Nonce.new()
    r = SEEC.gen_rand(seec_sk)
    ct_init, k_init = IKem.enc(pk_init, r)
    payload.add(ct_init)
    c = AEAD.enc(key_0, payload, h, nonce)
    nonce.increment()
    h = Hash(h, c)
    buffer.add(c)
    payload.flush()
    prekey_init, prekey_resp = HashObject.finalize(ck, k_init)
    key_init = AEAD.gen(prekey_init)
    key_resp = AEAD.gen(prekey_resp)
    nonce = AEAD.Nonce.new()
    payload.add(m)
    c = AEAD.enc(key_resp, payload, h, nonce)
    nonce.increment()
```

```
    buffer.add(c)
    h = Hash(h, c)
    return buffer

def receive_init_1(buffer):
    payload = buffer.parse_next()
    h = Hash(h, payload)
    ct_e = payload.parse_next()
    k_e = EKem.dec(sk_e, ct_e)
    prekey = ck.input(k_e)
    key_0 = AEAD.gen(prekey)
    nonce = AEAD.Nonce.new()
    c = buffer.parse_next()
    payload = AEAD.dec(key_0, c, h, nonce)
    h = Hash(h, c)
    nonce.increment()
    ct_init = payload.parse_next()
    k_init = IKem.dec(sk_init, ct_init)
    prekey_init, prekey_resp = HashObject.finalize(ck, k_init)
    key_init = AEAD.gen(prekey_init)
    key_resp = AEAD.gen(prekey_resp)
    nonce = AEAD.Nonce.new()
    c = buffer.parse_next()
    payload = AEAD.dec(key_resp, c, h, nonce)
    h = Hash(h, c)
    nonce.increment()
    m = payload.parse_next()
    return m
```

Listing B.12: PQIK

```
def keygen_init():
    seec_sk = SEEC.gen_key()
    pk_init, sk_init = IKem.gen()

def keygen_resp():
    seec_sk = SEEC.gen_key()
    pk_resp, sk_resp = RKem.gen()
    Publish(pk_resp)

def initialize_init():
    h = Hash("pqIK_label")
    ck = HashObject.gen("pqIK_label")

def send_init_1(m):
    payload = ""
    buffer = ""
    r = SEEC.gen_rand(seec_sk)
    ct_resp, k_resp = RKem.enc(pk_resp, r)
    payload.add(ct_resp)
```

```
    h = Hash(h, payload)
    buffer.add(payload)
    payload.flush()
    prekey = ck.input(k_resp)
    key_0 = AEAD.gen(prekey)
    nonce = AEAD.Nonce.new()
    r = SEEC.gen_rand(seec_sk)
    pk_e, sk_e = EKem.gen(r)
    payload.add(pk_e)
    payload.add(pk_init)
    payload.add(m)
    c = AEAD.enc(key_0, payload, h, nonce)
    nonce.increment()
    buffer.add(c)
    h = Hash(h, c)
    return buffer

def initialize_resp():
    h = Hash("pqIK_label")
    ck = HashObject.gen("pqIK_label")

def receive_resp_1(buffer):
    payload = buffer.parse_next()
    h = Hash(h, payload)
    ct_resp = payload.parse_next()
    k_resp = RKem.dec(sk_resp, ct_resp)
    prekey = ck.input(k_resp)
    key_0 = AEAD.gen(prekey)
    nonce = AEAD.Nonce.new()
    c = buffer.parse_next()
    payload = AEAD.dec(key_0, c, h, nonce)
    h = Hash(h, c)
    nonce.increment()
    pk_e = payload.parse_next()
    pk_init = payload.parse_next()
    m = payload.parse_next()
    return m

def send_resp_1(m):
    payload = ""
    buffer = ""
    r = SEEC.gen_rand(seec_sk)
    ct_e, k_e = EKem.enc(pk_e, r)
    payload.add(ct_e)
    h = Hash(h, payload)
    buffer.add(payload)
    payload.flush()
    prekey = ck.input(k_e)
    key_1 = AEAD.gen(prekey)
    nonce = AEAD.Nonce.new()
```

```
    r = SEEC.gen_rand(seec_sk)
    ct_init, k_init = IKem.enc(pk_init, r)
    payload.add(ct_init)
    c = AEAD.enc(key_1, payload, h, nonce)
    nonce.increment()
    h = Hash(h, c)
    buffer.add(c)
    payload.flush()
    prekey_init, prekey_resp = HashObject.finalize(ck, k_init)
    key_init = AEAD.gen(prekey_init)
    key_resp = AEAD.gen(prekey_resp)
    nonce = AEAD.Nonce.new()
    payload.add(m)
    c = AEAD.enc(key_resp, payload, h, nonce)
    nonce.increment()
    buffer.add(c)
    h = Hash(h, c)
    return buffer

def receive_init_1(buffer):
    c = buffer.parse_next()
    payload = AEAD.dec(key_0, c, h, nonce)
    h = Hash(h, c)
    nonce.increment()
    ct_e = payload.parse_next()
    k_e = EKem.dec(sk_e, ct_e)
    prekey = ck.input(k_e)
    key_1 = AEAD.gen(prekey)
    nonce = AEAD.Nonce.new()
    c = buffer.parse_next()
    payload = AEAD.dec(key_0, c, h, nonce)
    h = Hash(h, c)
    nonce.increment()
    ct_init = payload.parse_next()
    k_init = IKem.dec(sk_init, ct_init)
    prekey_init, prekey_resp = HashObject.finalize(ck, k_init)
    key_init = AEAD.gen(prekey_init)
    key_resp = AEAD.gen(prekey_resp)
    nonce = AEAD.Nonce.new()
    c = buffer.parse_next()
    payload = AEAD.dec(key_resp, c, h, nonce)
    h = Hash(h, c)
    nonce.increment()
    m = payload.parse_next()
    return m
```

Listing B.13: PQIX

```
def keygen_init():
    seec_sk = SEEC.gen_key()
```

```
    pk_init, sk_init = IKem.gen()

def keygen_resp():
    seec_sk = SEEC.gen_key()
    pk_resp, sk_resp = RKem.gen()

def initialize_init():
    h = Hash("pqIX_label")
    ck = HashObject.gen("pqIX_label")

def send_init_1(m):
    payload = ""
    buffer = ""
    r = SEEC.gen_rand(seec_sk)
    pk_e, sk_e = EKem.gen(r)
    payload.add(pk_e)
    payload.add(pk_init)
    payload.add(m)
    buffer.add(payload)
    h = Hash(h, payload)
    return buffer

def initialize_resp():
    h = Hash("pqIX_label")
    ck = HashObject.gen("pqIX_label")

def receive_resp_1(buffer):
    payload = buffer.parse_next()
    h = Hash(h, payload)
    pk_e = payload.parse_next()
    pk_init = payload.parse_next()
    m = payload.parse_next()
    return m

def send_resp_1(m):
    payload = ""
    buffer = ""
    r = SEEC.gen_rand(seec_sk)
    ct_e, k_e = EKem.enc(pk_e, r)
    payload.add(ct_e)
    h = Hash(h, payload)
    buffer.add(payload)
    payload.flush()
    prekey = ck.input(k_e)
    key_0 = AEAD.gen(prekey)
    nonce = AEAD.Nonce.new()
    r = SEEC.gen_rand(seec_sk)
    ct_init, k_init = IKem.enc(pk_init, r)
    payload.add(ct_init)
    c = AEAD.enc(key_0, payload, h, nonce)
```

```
    nonce.increment()
    h = Hash(h, c)
    buffer.add(c)
    payload.flush()
    prekey = ck.input(k_init)
    key_1 = AEAD.gen(prekey)
    nonce = AEAD.Nonce.new()
    payload.add(pk_resp)
    payload.add(m)
    c = AEAD.enc(key_1, payload, h, nonce)
    nonce.increment()
    buffer.add(c)
    h = Hash(h, c)
    return buffer

def receive_init_1(buffer):
    payload = buffer.parse_next()
    h = Hash(h, payload)
    ct_e = payload.parse_next()
    k_e = EKem.dec(sk_e, ct_e)
    prekey = ck.input(k_e)
    key_0 = AEAD.gen(prekey)
    nonce = AEAD.Nonce.new()
    c = buffer.parse_next()
    payload = AEAD.dec(key_0, c, h, nonce)
    h = Hash(h, c)
    nonce.increment()
    ct_init = payload.parse_next()
    k_init = IKem.dec(sk_init, ct_init)
    prekey = ck.input(k_init)
    key_1 = AEAD.gen(prekey)
    nonce = AEAD.Nonce.new()
    c = buffer.parse_next()
    payload = AEAD.dec(key_0, c, h, nonce)
    h = Hash(h, c)
    nonce.increment()
    pk_resp = payload.parse_next()
    m = payload.parse_next()
    return m

def send_init_2(m):
    payload = ""
    buffer = ""
    r = SEEC.gen_rand(seec_sk)
    ct_resp, k_resp = RKem.enc(pk_resp, r)
    payload.add(ct_resp)
    c = AEAD.enc(key_1, payload, h, nonce)
    nonce.increment()
    h = Hash(h, c)
    buffer.add(c)
```

```
    payload.flush()
    prekey_init, prekey_resp = HashObject.finalize(ck, k_resp)
    key_init = AEAD.gen(prekey_init)
    key_resp = AEAD.gen(prekey_resp)
    nonce = AEAD.Nonce.new()
    payload.add(m)
    c = AEAD.enc(key_init, payload, h, nonce)
    nonce.increment()
    buffer.add(c)
    h = Hash(h, c)
    return buffer

def receive_resp_2(buffer):
    c = buffer.parse_next()
    payload = AEAD.dec(key_1, c, h, nonce)
    h = Hash(h, c)
    nonce.increment()
    ct_resp = payload.parse_next()
    k_resp = RKem.dec(sk_resp, ct_resp)
    prekey_init, prekey_resp = HashObject.finalize(ck, k_resp)
    key_init = AEAD.gen(prekey_init)
    key_resp = AEAD.gen(prekey_resp)
    nonce = AEAD.Nonce.new()
    c = buffer.parse_next()
    payload = AEAD.dec(key_init, c, h, nonce)
    h = Hash(h, c)
    nonce.increment()
    m = payload.parse_next()
    return m
```

C. PQC in Space

C.1. Detailed KEM+Signature Protocol

The detailed protocol for the KEM+Signature-update contains protection against a power-outage; this is especially necessary for one-time-signatures as they must not be used to sign a different message, even if a handshake is interrupted.

The idea here is to derandomize the protocol by sampling a random seed before anything else and storing it in safe memory. In the event of a power-outage this seed will not be regenerated, but just loaded from storage. After the completion of the handshake it will on the other hand be deleted.

This seed is then fed into a cryptographically secure random-number-generator whose output is the sole randomness used in the exchange from that point onwards. If the satellite has a power-outage during the exchange it will restart the exchange from the beginning waiting for mission control to resend its initial packet. Mission control must resend the exact same packet until it arrives and the satellite can verify it. In the case of a signature-chain there is no risk of a different message arriving since the signing-key of mission control is only used once for the specific message that the satellite should receive. The satellite will then recompute its response deterministically and resend it, until mission control receives it.

In the case of a power-outage at mission control, the recreatable randomness allows to recreate the exact initial message and update in a similar manner.

We remark that this protection is possible in general, but possibly less relevant for KEM-based challenge-response protocols, since they can more easily be rerun independently in case of a power-failure.

```
def send_1():
    if seed is None: # Possibly set from aborted earlier interaction
        seed = true_rng()
    if ctr_mc is None:
        ctr_mc = 0
    prng = csprng(seed)
    prho_state_mc = NoiseHashObject.gen("")
```

```
    psk = (key_mc + key_sat) if key_mc is not None else b'\0\0...'
    h_0 = H(PROTOCOL_TOKEN, pk_mc, pk_sat)
    [k_0, _] = prho_state_mc.input(psk)
    pk_e, sk_e = EKEM.gen(prng())
    h_1 = H(h_0, pk_e)
    ctr_mc += 1
    payload = ctr_mc
    if update_longterm:
        pk_mc_new, sk_mc_new = Sig.gen(prng())
        payload += pk_mc_new
    c_0 = AEAD.enc(k_0, payload, 0, h_0)
    h_2 = H(h_1, c_0)
    sig_mc = Sig.sign(sk_mc, h_2) # optionally encrypt
    sk_mc = sk_mc_new # Optional: wait until response for key-confirmation
    h_3 = H(h_2, sig_mc)
    send(pk_e, c_0, sig_mc)

def receive_1((pk_e, c_0, sig_mc)):
    if seed is None: # Possibly set from aborted earlier interaction
        seed = true_rng()
    if ctr_sat is None:
        ctr_sat = 0
    prng = csprng(seed)
    prho_state_sat = NoiseHashObject.gen()
    psk = (key_mc + key_sat) if key_mc is not None else b'\0\0...'
    h_0 = H(PROTOCOL_TOKEN, pk_mc, pk_sat)
    [k_0, _] = prho_state_sat.input(psk)
    h_1 = H(h_0, pk_e)
    ctr_mc, pk_mc_new = AEAD.dec(k_0, c_0, 0, h_0)
    h_2 = H(h_1, c_0)
    if not Sig.verify(pk_mc, sig_mc, h2) or ctr_mc != ctr_sat + 1:
        error()
    ctr_sat = ctr_mc
    h_3 = H(h_2, sig_mc)
    if update_longterm:
        pk_mc = pk_mc_new

def send_2():
    c_e, k_e = EKEM.enc(prng())
    # c_e may optionally be encrypted with k_0
    h_4 = H(h_3, c_e)
    [k_1, _] = prho_state_sat.input(k_e)
    payload = ""
    if update_longterm:
        pk_sat_new, sk_sat_new = Sig.gen(prng())
        payload += pk_sat_new
    c_1 = AEAD.enc(k_1, payload, 0, h_4)
    h_5 = H(h_4, c_1)
    sig_sat = Sig.sign(sk_sat, h_5) # optionally encrypt
    sk_sat = sk_sat_new # Optional: wait until response for key-confirmation
```

```
58      h_6 = H(h_5, sig_sat)
59      send(c_e, c_1, sig_sat)
60      [pre_key_mc, pre_key_sat] = prho_state_sat.finalize("")
61      key_mc = KDF(pre_key_mc, h_6)
62      key_sat = KDF(pre_key_sat, h_6)
63      return key_mc, k_s
64
65  def receive_2((c_e, c_1, sig_sat)):
66      k_e = EKEM.dec(sk_e, c_e)
67      h_4 = H(h_3, c_e)
68      [k_1, _] = prho_state_mc.input(k_e)
69      pk_sat_new = AEAD.dec(k_1, c_1, 0, h_4)
70      h_5 = H(h_4, c_1)
71      h_6 = H(h_5, sig_sat)
72      if not Sig.verify(pk_sat, sig_sat, h_4):
73          error()
74      if update_longterm:
75          pk_sat = pk_sat_new
76      [pre_key_mc, pre_key_sat] = prho_state_mc.finalize("")
77      key_mc = KDF(pre_key_mc, h_6)
78      key_sat = KDF(pre_key_sat, h_6)
79      return key_mc, key_sat
```

C.2. Packet-Sizes

The following packet sizes assume the existence of `psk` and include the corresponding authentication tag.

- Triple-KEM:
  - Packet 1: RKEM-ct + AEAD-tag + EKEM-pk + IKEM-pk[opt1] + AEAD-tag
  - Packet 2: EKEM-ct + AEAD-tag + IKEM-ct + AEAD-tag + RKEM-pk[opt2] + AEAD-tag[opt2]
  - Packet 3: AEAD-tag
  - TK(Kyber512,McEliece348864)
    - Packet 1: 96 + 16 + 800 + 261120[opt1] + 16 = 928/262048
    - Packet 2: 768 + 16 +96 + 16 + 261120[opt2] + 16[opt2] = 896/262032
    - Packet 3: 16
  - TK(Kyber512+X25519)

    - Packet 1: 800 + 16 + 832 + 832[opt1] + 16 = 1664/2496
    - Packet 2: 800 + 16+ 800 + 16 + 832[opt2] + 16[opt2] = 1632/2480
    - Packet 3: 16
  - TK(Kyber768+X25519)
    - Packet 1: 1120 + 16 + 1216 + 1216[opt1] + 16 = 2368/3584
    - Packet 2: 1120 + 16 + 1120 + 16 + 1216[opt2] + 16[opt2] = 2272/3504
    - Packet 3: 16
  - TK(Kyber1024+X25519)
    - Packet 1: 1600 + 16 + 1600 + 1600[opt1] + 16 = 3232/4832
    - Packet 2: 1600 + 16 + 1600 + 16 + 1600[opt2] + 16[opt2] = 3232/4848
    - Packet 3: 16

- Sign+KEM:

  The following packet sizes assume the existence of `psk` and include the corresponding authentication tag.

  - Packet 1: ctr + EKEM-pk + Sig-pk[opt1] + AEAD-tag + Sig
  - Packet 2: EKEM-ct, Sig-pk[opt2] + AEAD-tag + Sig
  - SK(Kyber512+X25519, Dilithium2+ECDSA)
    - Packet 1: 16 + 832 + 1344[opt1] + 16 + 2484 =3348/4692
    - Packet 2: 800 + 1344[opt2] + 16 + 2484 = 3300/4644
  - SK(Kyber512+X25519, Falcon512+ECDSA)
    - Packet 1: 16 + 832 + 929[opt1] + 16 + 730 = 1594/2523
    - Packet 2: 800 + 929[opt2] + 16 + 730 = 1546/2475
  - SK(Kyber512+X25519, XMSS-SHA2_10_256)
    - Packet 1: 16 + 832 + 64[opt1] + 16 + 2500 = 3364/3428
    - Packet 2: 800 + 64[opt2] + 16 + 2500 = 3316/3380

- Signature-Chain
  - Packet 1: EKEM-pk + Sig-pk + AEAD-tag + Sig

- Packet 2: EKEM-ct, Sig-pk + AEAD-tag + Sig
- SC(Kyber512+X25519,WOTS+(32,16))
  - Packet 1: 832 + 32 + 16 + 2144 = 3024
  - Packet 2: 800 + 32 + 16 + 2144 = 2992
- SC(Kyber768+X25519,WOTS+(32,16))
  - Packet 1: 1216 + 32 + 16 + 2144 = 3408
  - Packet 2: 1120 + 32 + 16 + 2144 = 3312
- SC(Kyber1024+X25519,WOTS+(32,16))
  - Packet 1: 1600 + 32 + 16 + 2144 = 3792
  - Packet 2: 1600 + 32 + 16 + 2144 = 3792
- SC(Kyber1024+X25519,WOTS+(64,16))
  - Packet 1: 1600 + 32 + 16 + 8384 = 10032
  - Packet 2: 1600 + 32 + 16 + 8384 = 10032

Bibliography

[AAB+22] Carlos Aguilar-Melchor, Nicolas Aragon, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Edoardo Persichetti, Gilles Zémor, Jurjen Bos, Arnaud Dion, Jerome Lacan, Jean-Marc Robert, and Pascal Veron. HQC. Technical report, National Institute of Standards and Technology, 2022. available at https://csrc.nist.gov/Projects/post-quantum-cryptography/round-4-submissions. 3, 213

[AAC+22] Gorjan Alagic, Daniel Apon, David Cooper, Quynh Dang, Thinh Dang, John Kelsey, Jacob Lichtinger, Yi-Kai Liu, Carl Miller, Dustin Moody, Rene Peralta, Ray Perlner, Angela Robinson, and Daniel Smith-Tone. Status Report on the Third Round of the NIST Post-Quantum Cryptography Standardization Process. Technical report, Sept 2022. 211, 214

[AASA+19] Gorjan Alagic, Jacob Alperin-Sheriff, Daniel Apon, David Cooper, Quynh Dang, Yi-Kai Liu, Carl Miller, Dustin Moody, Rene Peralta, Ray Perlner, Angela Robinson, and Daniel Smith-Tone. Status report on the first round of the NIST post-quantum cryptography standardization process. NISTIR 8240, 2019. https://nvlpubs.nist.gov/nistpubs/ir/2019/NIST.IR.8240.pdf. 109

[ABB+22] Nicolas Aragon, Paulo Barreto, Slim Bettaieb, Loic Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Phillipe Gaborit, Shay Gueron, Tim Guneysu, Carlos Aguilar-Melchor, Rafael Misoczki, Edoardo Persichetti, Nicolas Sendrier, Jean-Pierre Tillich, Gilles Zémor, Valentin Vasseur, Santosh Ghosh, and Jan Richter-Brokmann. BIKE. Technical report, National Institute of Standards and Technology, 2022. available at https://csrc.nist.gov/Projects/post-quantum-cryptography/round-4-submissions. 3, 213

[ABC+22] Martin R. Albrecht, Daniel J. Bernstein, Tung Chou, Carlos Cid, Jan Gilcher, Tanja Lange, Varun Maram, Ingo von Maurich, Rafael

Misoczki, Ruben Niederhagen, Kenneth G. Paterson, Edoardo Persichetti, Christiane Peters, Peter Schwabe, Nicolas Sendrier, Jakub Szefer, Cen Jung Tjhai, Martin Tomlinson, and Wen Wang. Classic McEliece. Technical report, National Institute of Standards and Technology, 2022. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-4-submissions. 2, 3, 212

[ABD+21] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. Crystals-kyber (version 3.02) – submission to round 3 of the nist post-quantum project, 2021. https://pq-crystals.org/kyber/data/kyber-specification-round3-20210804.pdf. 198

[ACG+20] Liliya R. Akhmetzyanova, Cas Cremers, Luke Garratt, Stanislav Smyshlyaev, and Nick Sullivan. Limiting the impact of unreliable randomness in deployed security protocols. In Limin Jia and Ralf Küsters, editors, *CSF 2020: IEEE 33rd Computer Security Foundations Symposium*, pages 277–287, Boston, MA, USA, June 22–26, 2020. IEEE Computer Society Press. 165, 168

[ADH+22a] Yawning Angel, Benjamin Dowling, Andreas Hülsing, Peter Schwabe, and Fiona Johanna Weber. Post quantum noise. Cryptology ePrint Archive, Report 2022/539, 2022. 157

[ADH+22b] Yawning Angel, Benjamin Dowling, Andreas Hülsing, Peter Schwabe, and Fiona Johanna Weber. Post quantum noise. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022: 29th Conference on Computer and Communications Security*, pages 97–109, Los Angeles, CA, USA, November 7–11, 2022. ACM Press. 157, 160, 198, 201, 222, 224, 233, 242

[ADPS16] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. NewHope without reconciliation. Cryptology ePrint Archive, Report 2016/1157, 2016. 91

[AGGM06] Adi Akavia, Oded Goldreich, Shafi Goldwasser, and Dana Moshkovitz. On basing one-way functions on NP-hardness. In Jon M. Kleinberg, editor, *38th Annual ACM Symposium on Theory of Computing*, pages 701–710, Seattle, WA, USA, May 21–23, 2006. ACM Press. 4

[AMW19] Jacob Appelbaum, Chloe Martindale, and Peter Wu. Tiny WireGuard tweak. In Johannes Buchmann, Abderrahmane Nitaj, and Tajje eddine Rachidi, editors, *AFRICACRYPT 19: 11th International Conference on Cryptology in Africa*, volume 11627 of *Lecture Notes in Computer Science*, pages 3–20, Rabat, Morocco, July 9–11, 2019. Springer, Cham, Switzerland. 88, 101, 103, 104

[Ang] Yawning Angel. nyquist - a Noise protocol framework implementation. https://github.com/Yawning/nyquist. 198

[AN99] ANSI X9.62-1999. *Public key cryptography for the financial services industry: The elliptic curve digital signature algorithm (ECDSA)*, 1999. 222

[ANWW13] Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O'Hearn, and Christian Winnerlein. BLAKE2: Simpler, smaller, fast as MD5. In Michael J. Jacobson, Jr., Michael E. Locasto, Payman Mohassel, and Reihaneh Safavi-Naini, editors, *ACNS 13: 11th International Conference on Applied Cryptography and Network Security*, volume 7954 of *Lecture Notes in Computer Science*, pages 119–135, Banff, AB, Canada, June 25–28, 2013. Springer Berlin Heidelberg, Germany. 87

[Atl12] Antonios Atlasis. Attacking IPv6 implementation using fragmentation. Blackhat Europe, 2012. http://media.blackhat.com/bh-eu-12/Atlasis/bh-eu-12-Atlasis-Attacking_IPv6-WP.pdf. 90

[BBC+20] Daniel J. Bernstein, Billy Bob Brumley, Ming-Shing Chen, Chitchanok Chuengsatiansup, Tanja Lange, Adrian Marotzke, Bo-Yuan Peng, Nicola Tuveri, Christine van Vredendaal, and Bo-Yin Yang. NTRU Prime. Technical report, National Institute of Standards and Technology, 2020. available at https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions. 213

[BBC+21] Gustavo Banegas, Daniel J. Bernstein, Fabio Campos, Tung Chou, Tanja Lange, Michael Michael Meyer, Benjamin Smith, and Jana Sotáková. Ctidh: faster constant-time csidh. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, (4):351–387,

2021. https://tches.iacr.org/index.php/TCHES/article/view/9069. 159

[BBDP01] Mihir Bellare, Alexandra Boldyreva, Anand Desai, and David Pointcheval. Key-privacy in public-key encryption. In Colin Boyd, editor, *Advances in Cryptology – ASIACRYPT 2001*, volume 2248 of *Lecture Notes in Computer Science*, pages 566–582, Gold Coast, Australia, December 9–13, 2001. Springer Berlin Heidelberg, Germany. 107

[BBF+19] Hayo Baan, Sauvik Bhattacharya, Scott Fluhrer, Oscar Garcia-Morchon, Thijs Laarhoven, Rachel Player, Ronald Rietman, Markku-Juhani O. Saarinen, Ludo Tolhuizen, Jos e Luis Torre-Arce, , and Zhenfei Zhang. Round5: KEM and PKE based on (ring) learning with rounding. Round-2 submission to the NIST PQC project, 2019. https://round5.org/#spec. 109

[BBR+23] Richard Barnes, Benjamin Beurdouche, Raphael Robert, Jon Millican, Emad Omara, and Katriel Cohn-Gordon. The Messaging Layer Security (MLS) Protocol. RFC 9420, July 2023. 6, 27, 28, 53

[BCD+16] Joppe W Bos, Craig Costello, Léo Ducas, Ilya Mironov, Michael Naehrig, Valeria Nikolaenko, Ananth Raghunathan, and Douglas Stebila. Frodo: Take off the ring! practical, quantum-secure key exchange from LWE. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, CCS'16*, pages 1006–1018. ACM, 2016. https://dl.acm.org/doi/10.1145/2976749.2978425. 112

[BCD+24] Manuel Barbosa, Deirdre Connolly, João Diogo Duarte, Aaron Kaiser, Peter Schwabe, Karoline Varner, and Bas Westerbaan. X-wing. *IACR Communications in Cryptology (CiC)*, 1(1):21, 2024. 221

[BCGP08] Colin Boyd, Yvonne Cliff, Juan González Nieto, and Kenneth G. Paterson. Efficient one-round key exchange in the standard model. In Yi Mu, Willy Susilo, and Jennifer Seberry, editors, *ACISP 08: 13th Australasian Conference on Information Security and Privacy*, volume 5107 of *Lecture Notes in Computer Science*, pages 69–83, Wollongong, Australia, July 7–9, 2008. Springer Berlin Heidelberg, Germany. 91

[BCL+19] Daniel J. Bernstein, Tung Chou, Tanja Lange, Ingo von Maurich, Rafael Misoczki, Ruben Niederhagen, Edoardo Persichetti, Christiane Peters, Peter Schwabe, Nicolas Sendrier, Jakub Szefer, and Wen Wang. Classic McEliece: conservative code-based cryptography. Round-2 submission to the NIST PQC project, 2019. https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/round-2/submissions/Classic-McEliece-Round2.zip. 90, 107

[BDF+11] Dan Boneh, Özgür Dagdelen, Marc Fischlin, Anja Lehmann, Christian Schaffner, and Mark Zhandry. Random oracles in a quantum world. In Dong Hoon Lee and Xiaoyun Wang, editors, *Advances in Cryptology – ASIACRYPT 2011*, volume 7073 of *Lecture Notes in Computer Science*, pages 41–69, Seoul, South Korea, December 4–8, 2011. Springer Berlin Heidelberg, Germany. 5

[BDH11] Johannes A. Buchmann, Erik Dahmen, and Andreas Hülsing. XMSS - A practical forward secure signature scheme based on minimal security assumptions. In Bo-Yin Yang, editor, *Post-Quantum Cryptography - 4th International Workshop, PQCrypto 2011*, pages 117–129, Tapei, Taiwan, November 29 – December 2 2011. Springer Berlin Heidelberg, Germany. 59

[BDH+21] Ward Beullens, Jan-Pieter D'Anvers, Andreas Hülsing, Tanja Lange, Lorenz Panny, Cyprien de Saint Guilhem, Nigel P. Smart, Evangelos Rekleitis, Angeliki Aktypi, and Athanasios-Vasileios Grammatopoulos. Post-quantum cryptography: Current state and quantum mitigation, 2021. https://www.enisa.europa.eu/publications/post-quantum-cryptography-current-state-and-quantum-mitigation/@@download/fullReport. 211

[BDK+18] Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, and Damien Stehlé. CRYSTALS – Kyber: a CCA-secure module-lattice-based KEM. In *2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018*, pages 353–367. IEEE, 2018. https://cryptojedi.org/papers/#kyber. 92, 198

[BDL+11] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. In Bart Preneel and Tsuyoshi Takagi, editors, *Cryptographic Hardware and Embedded*

Systems – CHES 2011, volume 6917 of *Lecture Notes in Computer Science*, pages 124–142, Nara, Japan, September 28 – October 1, 2011. Springer Berlin Heidelberg, Germany. 222

[Ber05] Daniel J. Bernstein. The Poly1305-AES message-authentication code. In Henri Gilbert and Helena Handschuh, editors, *Fast Software Encryption*, volume 3557 of *LNCS*, pages 32–49. Springer, 2005. http://cr.yp.to/papers.html#poly1305. 87

[Ber06] Daniel J. Bernstein. Curve25519: New Diffie-Hellman speed records. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *PKC 2006: 9th International Conference on Theory and Practice of Public Key Cryptography*, volume 3958 of *Lecture Notes in Computer Science*, pages 207–228, New York, NY, USA, April 24–26, 2006. Springer Berlin Heidelberg, Germany. 87, 159

[Ber08] Daniel J. Bernstein. ChaCha, a variant of Salsa20. In *Workshop Record of SASC 2008: The State of the Art of Stream Ciphers*, 2008. http://cr.yp.to/papers.html#chacha. 87

[Ber19a] Daniel J. Bernstein. Re: [pqc-forum] new quantum cryptanalysis of CSIDH. Posting to the NIST pqc-forum mailing list, 2019. https://groups.google.com/a/list.nist.gov/forum/#!original/pqc-forum/svm1kDy6c54/0gFOLitbAgAJ. 89

[Ber19b] Daniel J. Bernstein. Re: [pqc-forum] ROUND 2 OFFICIAL COMMENT: NewHope. Posting to the NIST pqc-forum mailing list, 2019. https://groups.google.com/a/list.nist.gov/forum/#!original/pqc-forum/u3FoYrN-7fk/3EZwDIvDBQAJ. 91

[Ber19c] Daniel J. Bernstein. Re: [pqc-forum] ROUND 2 OFFICIAL COMMENT: NewHope. Posting to the NIST pqc-forum mailing list, 2019. https://groups.google.com/a/list.nist.gov/forum/#!original/pqc-forum/u3FoYrN-7fk/MxBVn9M7CQAJ. 91

[Beu22] Ward Beullens. Breaking rainbow takes a weekend on a laptop. In Yevgeniy Dodis and Thomas Shrimpton, editors, *Advances in Cryptology – CRYPTO 2022, Part II*, volume 13508 of *Lecture Notes in Computer Science*, pages 464–479, Santa Barbara, CA, USA, August 15–18, 2022. Springer, Cham, Switzerland. 217

[BFGJ17] Jacqueline Brendel, Marc Fischlin, Felix Günther, and Christian Janson. PRF-ODH: Relations, instantiations, and impossibility results. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology – CRYPTO 2017, Part III*, volume 10403 of *Lecture Notes in Computer Science*, pages 651–681, Santa Barbara, CA, USA, August 20–24, 2017. Springer, Cham, Switzerland. 11

[BGB04] Nikita Borisov, Ian Goldberg, and Eric Brewer. Off-the-record communication, or, why not to use pgp. In *2004 ACM Workshop on Privacy in the Electronic Society*, WPES '04, page 77–84, New York, NY, USA, 2004. Association for Computing Machinery. 28

[BGJ+15] Nir Bitansky, Shafi Goldwasser, Abhishek Jain, Omer Paneth, Vinod Vaikuntanathan, and Brent Waters. Time-lock puzzles from randomized encodings. Cryptology ePrint Archive, Report 2015/514, 2015. 61

[BH23] Nina Bindel and Britta Hale. A note on hybrid signature schemes. Cryptology ePrint Archive, Paper 2023/423, 2023. https://eprint.iacr.org/2023/423. 222

[BHLR22] Daniel J. Bernstein, Andreas Hülsing, Tanja Lange, and Evangelos Rekleitis. Post-quantum cryptography - integration study, 2022. https://www.enisa.europa.eu/publications/post-quantum-cryptography-integration-study. 220

[BL] Daniel J. Bernstein and Tanja Lange. eBACS: ECRYPT benchmarking of cryptographic systems. https://bench.cr.yp.to (accessed 29 Sep 2021). 110, 198

[Bla] Bruno Blanchet. ProVerif: Cryptographic protocol verifier in the formal model. https://prosecco.gforge.inria.fr/personal/bblanche/proverif/. 92

[BLMP19] Daniel J. Bernstein, Tanja Lange, Chloe Martindale, and Lorenz Panny. Quantum circuits for the CSIDH: Optimizing quantum evaluation of isogenies. In Yuval Ishai and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2019, Part II*, volume 11477 of *Lecture Notes in Computer Science*, pages 409–441, Darmstadt, Germany, May 19–23, 2019. Springer, Cham, Switzerland. 89, 159

[BLN16] Daniel J Bernstein, Tanja Lange, and Ruben Niederhagen. Dual ec: A standardized back door. In *The New Codebreakers*, pages 256–281. Springer, 2016. 164

[BM99] Mihir Bellare and Sara K. Miner. A forward-secure digital signature scheme. In Michael J. Wiener, editor, *Advances in Cryptology – CRYPTO'99*, volume 1666 of *Lecture Notes in Computer Science*, pages 431–448, Santa Barbara, CA, USA, August 15–19, 1999. Springer Berlin Heidelberg, Germany. 59

[Bor] BoringTun. https://github.com/cloudflare/boringtun. 88

[BR93] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In Dorothy E. Denning, Raymond Pyle, Ravi Ganesan, Ravi S. Sandhu, and Victoria Ashby, editors, *ACM CCS 93: 1st Conference on Computer and Communications Security*, pages 62–73, Fairfax, Virginia, USA, November 3–5, 1993. ACM Press. 5

[BR94] Mihir Bellare and Phillip Rogaway. Entity authentication and key distribution. In Douglas R. Stinson, editor, *Advances in Cryptology – CRYPTO'93*, volume 773 of *Lecture Notes in Computer Science*, pages 232–249, Santa Barbara, CA, USA, August 22–26, 1994. Springer Berlin Heidelberg, Germany. 240

[BS20] Xavier Bonnetain and André Schrottenloher. Quantum security analysis of CSIDH. In Anne Canteaut and Yuval Ishai, editors, *Advances in Cryptology – EUROCRYPT 2020, Part II*, volume 12106 of *Lecture Notes in Computer Science*, pages 493–522, Zagreb, Croatia, May 10–14, 2020. Springer, Cham, Switzerland. 89, 159

[BST07] J. Bian, R. Seker, and U. Topaloglu. Off-the-record instant messaging for group conversation. In *2007 IEEE International Conference on Information Reuse and Integration*, pages 79–84, 2007. 29

[CD22] Wouter Castryck and Thomas Decru. An efficient key recovery attack on SIDH (preliminary version). Cryptology ePrint Archive, Report 2022/975, 2022. 90

[CD23] Wouter Castryck and Thomas Decru. An efficient key recovery attack on SIDH. In Carmit Hazay and Martijn Stam, editors, *Advances in Cryptology – EUROCRYPT 2023, Part V*, volume 14008 of *Lecture Notes in Computer Science*, pages 423–447, Lyon, France, April 23–27, 2023. Springer, Cham, Switzerland. 213, 214

[CDH+20] Cong Chen, Oussama Danba, Jeffrey Hoffstein, Andreas Hulsing, Joost Rijneveld, John M. Schanck, Peter Schwabe, William

Whyte, Zhenfei Zhang, Tsunekazu Saito, Takashi Yamakawa, and Keita Xagawa. NTRU. Technical report, National Institute of Standards and Technology, 2020. available at https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions. 212

[CDNO97] Ran Canetti, Cynthia Dwork, Moni Naor, and Rafail Ostrovsky. Deniable encryption. In Burton S. Kaliski, Jr., editor, *Advances in Cryptology – CRYPTO'97*, volume 1294 of *Lecture Notes in Computer Science*, pages 90–104, Santa Barbara, CA, USA, August 17–21, 1997. Springer Berlin Heidelberg, Germany. 28

[CF12] Cas J. F. Cremers and Michele Feltz. Beyond eCK: Perfect forward secrecy under actor compromise and ephemeral-key reveal. In Sara Foresti, Moti Yung, and Fabio Martinelli, editors, *ESORICS 2012: 17th European Symposium on Research in Computer Security*, volume 7459 of *Lecture Notes in Computer Science*, pages 734–751, Pisa, Italy, September 10–12, 2012. Springer Berlin Heidelberg, Germany. 259, 265, 266, 267

[CF15] Cas Cremers and Michèle Feltz. Beyond eCK: perfect forward secrecy under actor compromise and ephemeral-key reveal. *Designs, Codes and Cryptography*, 74(1):183–218, 2015. 96

[CFM+20] Antoine Casanova, Jean-Charles Faugère, Gilles Macario-Rat, Jacques Patarin, Ludovic Perret, and Jocelyn Ryckeghem. GeMSS. Technical report, National Institute of Standards and Technology, 2020. available at https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions. 218

[CGS+20] Cas Cremers, Luke Garratt, Stanislav V. Smyshlyaev, Nick Sullivan, and Christopher A. Wood. Randomness Improvements for Security Protocols. RFC 8937, October 2020. 165

[CHK19] Cas Cremers, Britta Hale, and Konrad Kohbrok. Efficient post-compromise security beyond one group. Cryptology ePrint Archive, Report 2019/477, 2019. 29

[CK01] Ran Canetti and Hugo Krawczyk. Analysis of key-exchange protocols and their use for building secure channels. In Birgit Pfitzmann, editor,

Advances in Cryptology – EUROCRYPT 2001, volume 2045 of *Lecture Notes in Computer Science*, pages 453–474, Innsbruck, Austria, May 6–10, 2001. Springer Berlin Heidelberg, Germany. 158

[CLM+18] Wouter Castryck, Tanja Lange, Chloe Martindale, Lorenz Panny, and Joost Renes. CSIDH: An efficient post-quantum commutative group action. In Thomas Peyrin and Steven Galbraith, editors, *Advances in Cryptology – ASIACRYPT 2018, Part III*, volume 11274 of *Lecture Notes in Computer Science*, pages 395–427, Brisbane, Queensland, Australia, December 2–6, 2018. Springer, Cham, Switzerland. 10, 89, 159

[Cou06] Jean-Marc Couveignes. Hard homogeneous spaces. Cryptology ePrint Archive, Report 2006/291, 2006. 10

[CPS19] Eric Crockett, Christian Paquin, and Douglas Stebila. Prototyping post-quantum and hybrid key exchange and authentication in TLS and SSH. Cryptology ePrint Archive, Report 2019/858, 2019. 222

[CV90] David Chaum and Hans Van Antwerpen. Undeniable signatures. In Gilles Brassard, editor, *Advances in Cryptology – CRYPTO'89*, volume 435 of *Lecture Notes in Computer Science*, pages 212–216, Santa Barbara, CA, USA, August 20–24, 1990. Springer, New York, USA. 28

[DCP+20] Jintai Ding, Ming-Shing Chen, Albrecht Petzoldt, Dieter Schmidt, Bo-Yin Yang, Matthias J. Kannwischer, and Jacques Patarin. Rainbow. Technical report, National Institute of Standards and Technology, 2020. available at https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions. 217

[Den03] Alexander W. Dent. A designer's guide to KEMs. In Kenneth G. Paterson, editor, *9th IMA International Conference on Cryptography and Coding*, volume 2898 of *Lecture Notes in Computer Science*, pages 133–151, Cirencester, UK, December 16–18, 2003. Springer Berlin Heidelberg, Germany. 11

[DH76] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976. 10, 157

[DH17] Steve Deering and Robert Hinden. Internet protocol, version 6 (IPv6) specification. IETF RFC 8200, 2017. https://tools.ietf.org/pdf/rfc8200.pdf. 90, 108

[DKR+20] Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, Frederik Vercauteren, Jose Maria Bermudo Mera, Michiel Van Beirendonck, and Andrea Basso. SABER. Technical report, National Institute of Standards and Technology, 2020. available at https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions. 212

[DKRV18] Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, and Frederik Vercauteren. Saber: Module-LWR based key exchange, CPA-secure encryption and CCA-secure KEM. In Antoine Joux, Abderrahmane Nitaj, and Tajjeeddine Rachidi, editors, *Progress in Cryptology – AFRICACRYPT 2018*, volume 10831 of *LNCS*, pages 282–305. Springer, 2018. https://eprint.iacr.org/2018/230. 90, 107

[DKRV19] Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, and Frederik Vercauteren. SABER: Mod-LWR based KEM (round 2 submission). Round-2 submission to the NIST PQC project, 2019. https://www.esat.kuleuven.be/cosic/pqcrypto/saber/resources.html. 90, 107

[DKS18] Luca De Feo, Jean Kieffer, and Benjamin Smith. Towards practical key exchange from ordinary isogeny graphs. In Thomas Peyrin and Steven Galbraith, editors, *Advances in Cryptology – ASIACRYPT 2018, Part III*, volume 11274 of *Lecture Notes in Computer Science*, pages 365–394, Brisbane, Queensland, Australia, December 2–6, 2018. Springer, Cham, Switzerland. 10

[DKSW09] Yevgeniy Dodis, Jonathan Katz, Adam Smith, and Shabsi Walfish. Composability and on-line deniability of authentication. In Omer Reingold, editor, *TCC 2009: 6th Theory of Cryptography Conference*, volume 5444 of *Lecture Notes in Computer Science*, pages 146–162. Springer Berlin Heidelberg, Germany, March 15–17, 2009. 28, 29, 31

[DM18] Jason Donenfeld and Kevin Milner. Formal verification of the WireGuard protocol, 2018. version June 7, 2018, https://www.

wireguard.com/papers/wireguard-formal-verification.pdf. 87, 89

[Don17] Jason A. Donenfeld. WireGuard: Next generation kernel network tunnel. In *ISOC Network and Distributed System Security Symposium – NDSS 2017*, San Diego, CA, USA, February 26 – March 1, 2017. The Internet Society. 87, 88, 96, 259

[DP08] The Debian-Project. Debian Security Advisory – DSA-1571-1 openssl – predictable random number generator, May 2008. https://www.debian.org/security/2008/dsa-1571. 164

[DP18] Benjamin Dowling and Kenneth G. Paterson. A cryptographic analysis of the WireGuard protocol. In Bart Preneel and Frederik Vercauteren, editors, *ACNS 18: 16th International Conference on Applied Cryptography and Network Security*, volume 10892 of *Lecture Notes in Computer Science*, pages 3–21, Leuven, Belgium, July 2–4, 2018. Springer, Cham, Switzerland. 87, 89, 96, 101, 104, 112, 176, 259

[DRS19] Benjamin Dowling, Paul Rösler, and Jörg Schwenk. Flexible authenticated and confidential channel establishment (fACCE): Analyzing the noise protocol framework. Cryptology ePrint Archive, Report 2019/436, 2019. 92

[DRS20] Benjamin Dowling, Paul Rösler, and Jörg Schwenk. Flexible authenticated and confidential channel establishment (fACCE): Analyzing the noise protocol framework. In Aggelos Kiayias, Markulf Kohlweiss, Petros Wallden, and Vassilis Zikas, editors, *PKC 2020: 23rd International Conference on Theory and Practice of Public Key Cryptography, Part I*, volume 12110 of *Lecture Notes in Computer Science*, pages 341–373, Edinburgh, UK, May 4–7, 2020. Springer, Cham, Switzerland. 158, 160, 164, 170, 171, 173, 198, 200, 269

[dV16] Simon de Vries. Achieving 128-bit security against quantum attacks in OpenVPN. Master's thesis, University of Twente, 2016. https://essay.utwente.nl/70677/1/2016-08-09%20MSc%20Thesis%20Simon%20de%20Vries%20final%20color.pdf. 92

[EKL+19] Karen Easterbrook, Kevin Kane, Brian LaMacchia, Dan Shumow, and Greg Zaverucha. Post-quantum cryptography VPN, 2019. https://www.microsoft.com/en-us/research/project/post-quantum-crypto-vpn/. 92, 110

[FHKP12] Eduarda S. V. Freire, Dennis Hofheinz, Eike Kiltz, and Kenneth G. Paterson. Non-interactive key exchange. Cryptology ePrint Archive, Report 2012/732, 2012. 159

[FO99] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In Michael J. Wiener, editor, *Advances in Cryptology – CRYPTO'99*, volume 1666 of *Lecture Notes in Computer Science*, pages 537–554, Santa Barbara, CA, USA, August 15–19, 1999. Springer Berlin Heidelberg, Germany. 91, 110

[FS12] Atsushi Fujioka and Koutarou Suzuki. Sufficient condition for identity-based authenticated key exchange resilient to leakage of secret keys. In Howon Kim, editor, *ICISC 11: 14th International Conference on Information Security and Cryptology*, volume 7259 of *Lecture Notes in Computer Science*, pages 490–509, Seoul, Korea, November 30 – December 2, 2012. Springer Berlin Heidelberg, Germany. 94

[fSaT16] National Institute for Standards and Technology. Submission requirements and evaluation criteria for the post-quantum cryptography standardization process, 2016. https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf. 108

[FSXY12] Atsushi Fujioka, Koutarou Suzuki, Keita Xagawa, and Kazuki Yoneyama. Strongly secure authenticated key exchange from factoring, codes, and lattices. In Marc Fischlin, Johannes Buchmann, and Mark Manulis, editors, *PKC 2012: 15th International Conference on Theory and Practice of Public Key Cryptography*, volume 7293 of *Lecture Notes in Computer Science*, pages 467–484, Darmstadt, Germany, May 21–23, 2012. Springer Berlin Heidelberg, Germany. 89, 91, 94, 99, 165

[FSXY15] Atsushi Fujioka, Koutarou Suzuki, Keita Xagawa, and Kazuki Yoneyama. Strongly secure authenticated key exchange from factoring, codes, and lattices. *Designs, Codes and Cryptography*, 76(3):469–504, 2015. 100

[FTTY19] Atsushi Fujioka, Katsuyuki Takashima, Shintaro Terada, and Kazuki Yoneyama. Supersingular isogeny Diffie-Hellman authenticated key exchange. In Kwangsu Lee, editor, *ICISC 18: 21st International Conference on Information Security and Cryptology*, volume

11396 of *Lecture Notes in Computer Science*, pages 177–195, Seoul, Korea, November 28–30, 2019. Springer, Cham, Switzerland. 92

[GdKQ+23] Phillip Gajland, Bor de Kock, Miguel Quaresma, Giulio Malavolta, and Peter Schwabe. Swoosh: Practical lattice-based non-interactive key exchange. Cryptology ePrint Archive, Report 2023/271, 2023. 10

[GHP13] Yoel Gluck, Neal Harris, and Angelo Prado. Breach: reviving the crime attack. *Unpublished manuscript*, 2013. 3

[GHP18] Federico Giacon, Felix Heuer, and Bertram Poettering. KEM combiners. In Michel Abdalla and Ricardo Dahab, editors, *PKC 2018: 21st International Conference on Theory and Practice of Public Key Cryptography, Part I*, volume 10769 of *Lecture Notes in Computer Science*, pages 190–218, Rio de Janeiro, Brazil, March 25–29, 2018. Springer, Cham, Switzerland. 220

[GHS+20] Guillaume Girol, Lucca Hirschi, Ralf Sasse, Dennis Jackson, Cas Cremers, and David Basin. A spectral analysis of noise: a comprehensive, automated, formal analysis of diffie-hellman protocols. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pages 1857–1874, 2020. 159

[GMR88] Shafi Goldwasser, Silvio Micali, and Ronald L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing*, 17(2):281–308, April 1988. 16

[GUVC09] Ian Goldberg, Berkant Ustaoglu, Matthew Van Gundy, and Hao Chen. Multi-party off-the-record messaging. In Ehab Al-Shaer, Somesh Jha, and Angelos D. Keromytis, editors, *ACM CCS 2009: 16th Conference on Computer and Communications Security*, pages 358–368, Chicago, Illinois, USA, November 9–13, 2009. ACM Press. 29

[HBD+22] Andreas Hülsing, Daniel J. Bernstein, Christoph Dobraunig, Maria Eichlseder, Scott Fluhrer, Stefan-Lukas Gazdag, Panos Kampanakis, Stefan Kölbl, Tanja Lange, Martin M. Lauridsen, Florian Mendel, Ruben Niederhagen, Christian Rechberger, Joost Rijneveld, Peter Schwabe, Jean-Philippe Aumasson, Bas Westerbaan, and Ward Beullens. SPHINCS+. Technical report, National Institute of Standards and Technol-

ogy, 2022. available at https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022. 2, 215

[HBG+18] Andreas Hülsing, Denise Butin, Stefan-Lukas Gazdag, Joost Rijneveld, and Aziz Mohaisen. XMSS: eXtended Merkle Signature Scheme. RFC 8391, RFC Editor, 05 2018. 3, 217

[HHK17] Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz. A modular analysis of the Fujisaki-Okamoto transformation. In Yael Kalai and Leonid Reyzin, editors, *TCC 2017: 15th Theory of Cryptography Conference, Part I*, volume 10677 of *Lecture Notes in Computer Science*, pages 341–371, Baltimore, MD, USA, November 12–15, 2017. Springer, Cham, Switzerland. 212

[Hin93] Harry Hinsley. The influence of ULTRA in the second world war, October 1993. transcript of a talk at Cambridge. 1

[HKS] Andreas Hülsing, Matthias Kannwischer, and Peter Schwabe. Forward-secure XMSS based on RFC 8391. https://github.com/mkannwischer/xmssfs. 78

[HKSU18] Kathrin Hövelmanns, Eike Kiltz, Sven Schäge, and Dominique Unruh. Generic authenticated key exchange in the quantum random oracle model. Cryptology ePrint Archive, Report 2018/928, 2018. 91

[HNS+20] Andreas Hülsing, Kai-Chun Ning, Peter Schwabe, Fiona Johanna Weber, and Philip R. Zimmermann. Post-quantum WireGuard. Cryptology ePrint Archive, Report 2020/379, 2020. 87

[HNS+21] Andreas Hülsing, Kai-Chun Ning, Peter Schwabe, Fiona Johanna Weber, and Philip R. Zimmermann. Post-quantum WireGuard. In *2021 IEEE Symposium on Security and Privacy*, pages 304–321, San Francisco, CA, USA, May 24–27, 2021. IEEE Computer Society Press. 87, 89, 109, 110, 111, 112, 159, 162, 176, 222, 224

[HW20] Andreas Hülsing and Fiona Johanna Weber. Epochal signatures for deniable group chats. Cryptology ePrint Archive, Report 2020/1138, 2020. 27

[HW21] Andreas Hülsing and Fiona Johanna Weber. Epochal signatures for deniable group chats. In *2021 IEEE Symposium on Security and Privacy*, pages 1677–1695, San Francisco, CA, USA, May 24–27, 2021. IEEE Computer Society Press. 27

[Imp95] Russell Impagliazzo. A personal view of average-case complexity. In *Proceedings of Structure in Complexity Theory. Tenth Annual IEEE Conference*, pages 134–147, 1995. 4

[Inc19] OpenVPN Inc. VPN software solutions & services for business | OpenVPN, 2019. https://openvpn.net/ (accessed 2019-10-21). 92

[ISE22] ISE Crypto PQC working group. Securing tomorrow today: Why Google now protects its internal communications from quantum threats, 2022. https://cloud.google.com/blog/products/identity-security/why-google-now-uses-post-quantum-cryptography-for-internal-comms. 213

[JAC+19] David Jao, Reza Azarderakhsh, Matthew Campagna, Craig Costello, Luca De Feo, Basil Hess, Amir Jalali, Brian Koziel, Brian LaMacchia, Patrick Longa, Michael Naehrig, Geovandro Pereira, Joost Renes, Vladimir Soukharev, and David Urbanik. Supersingular isogeny key encapsulation. Round-2 submission to the NIST PQC project, 2019. https://sike.org/#resources. 90, 109

[JAC+22] David Jao, Reza Azarderakhsh, Matthew Campagna, Craig Costello, Luca De Feo, Basil Hess, Amir Jalali, Brian Koziel, Brian LaMacchia, Patrick Longa, Michael Naehrig, Joost Renes, Vladimir Soukharev, David Urbanik, Geovandro Pereira, Koray Karabina, and Aaron Hutchinson. SIKE. Technical report, National Institute of Standards and Technology, 2022. available at https://csrc.nist.gov/Projects/post-quantum-cryptography/round-4-submissions. 3

[JKSS12] Tibor Jager, Florian Kohlar, Sven Schäge, and Jörg Schwenk. On the security of TLS-DHE in the standard model. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 273–293, Santa Barbara, CA, USA, August 19–23, 2012. Springer Berlin Heidelberg, Germany. 92

[JKSS17] Tibor Jager, Florian Kohlar, Sven Schäge, and Jörg Schwenk. Authenticated confidential channel establishment and the security of TLS-DHE. *Journal of Cryptology*, 30(4):1276–1324, October 2017. 158

[KFH19] Kris Kwiatkowski and Armando Faz-Hernández. Introducing circl: An advanced cryptographic library. Posting in the Cloudflare Blog, 2019. https://blog.cloudflare.com/introducing-circl/. 198

[KNB19] Nadim Kobeissi, Georgio Nicolas, and Karthikeyan Bhargavan. Noise Explorer: Fully automated modeling and verification for arbitrary Noise protocols. In *IEEE European Symposium on Security and Privacy (EuroS&P'19)*, pages 356–370. IEEE, 2019. https://eprint.iacr.org/2018/766. 92, 159

[KR98] Hugo Krawczyk and Tal Rabin. Chameleon hashing and signatures. Cryptology ePrint Archive, Report 1998/010, 1998. 28

[Kra05] Hugo Krawczyk. HMQV: A high-performance secure Diffie-Hellman protocol. In Victor Shoup, editor, *Advances in Cryptology – CRYPTO 2005*, volume 3621 of *Lecture Notes in Computer Science*, pages 546–566, Santa Barbara, CA, USA, August 14–18, 2005. Springer Berlin Heidelberg, Germany. 94, 158, 164

[KSW22] Stavros Kousidis, Falko Strenzke, and Aron Wussler. Post-quantum cryptography in openpgp draft-wussler-openpgp-pqc-00, 12 2022. https://datatracker.ietf.org/doc/draft-wussler-openpgp-pqc/. 221

[Lam81] Leslie Lamport. Password authentification with insecure communication. *Communications of the Association for Computing Machinery*, 24(11):770–772, December 1981. 59

[Lan13] Adam Langley. Hash based signatures. July 2013. https://www.imperialviolet.org/2013/07/18/hashsig.html. 3

[LBB19] Benjamin Lipp, Bruno Blanchet, and Karthikeyan Bhargavan. A mechanised cryptographic proof of the WireGuard virtual private network protocol. In *2019 IEEE European Symposium on Security and Privacy*, pages 231–246, Stockholm, Sweden, June 17–19, 2019. IEEE Computer Society Press. 92

[LDK+22] Vadim Lyubashevsky, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Peter Schwabe, Gregor Seiler, Damien Stehlé, and Shi Bai. CRYSTALS-DILITHIUM. Technical report, National Institute of Standards and Technology, 2022. available at https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022. 2, 215

[LLM07] Brian A. LaMacchia, Kristin Lauter, and Anton Mityagin. Stronger security of authenticated key exchange. In Willy Susilo, Joseph K. Liu, and Yi Mu, editors, *ProvSec 2007: 1st International Conference on Provable Security*, volume 4784 of *Lecture Notes in Computer Science*, pages 1–16, Wollongong, Australia, November 1–2, 2007. Springer Berlin Heidelberg, Germany. 96, 158, 165, 176, 240, 259

[Lon18] Patrick Longa. A note on post-quantum authenticated key exchange from supersingular isogenies. Cryptology ePrint Archive, Report 2018/267, 2018. 92

[LPR10] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In Henri Gilbert, editor, *Advances in Cryptology – EUROCRYPT 2010*, volume 6110 of *Lecture Notes in Computer Science*, pages 1–23, French Riviera, May 30 – June 3, 2010. Springer Berlin Heidelberg, Germany. 212

[LVH13] Hong Liu, Eugene Y. Vasserman, and Nicholas Hopper. Improved group off-the-record messaging. In *12th ACM Workshop on Workshop on Privacy in the Electronic Society*, WPES '13, page 249–254, New York, NY, USA, 2013. Association for Computing Machinery. 28

[MAB+19] Carlos Aguilar Melchor, Nicolas Aragon, Magali Bardet, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Adrien Hauteville, Ayoub Otmani, Olivier Ruatta, Jean-Pierre Tillich, and Gilles Zémor. ROLLO – Rank-Ouroboros, LAKE & LOCKER. Round-2 submission to the NIST PQC project, 2019. https://pqc-rollo.org/documentation.html. 109

[Mar13] Moxie Marlinspike. Advanced cryptographic ratcheting. 2013. https://signal.org/blog/advanced-ratcheting/. 28

[Mar14] Moxie Marlinspike. Private group messaging. 2014. https://signal.org/blog/private-groups/. 28

[MCF19] David A. McGrew, Michael Curcio, and Scott R. Fluhrer. Hash-Based Signatures. RFC 8554, RFC Editor, 2019. 3, 217

[MMP+23] Luciano Maino, Chloe Martindale, Lorenz Panny, Giacomo Pope, and Benjamin Wesolowski. A direct key recovery attack on SIDH. In Carmit Hazay and Martijn Stam, editors, *Advances in Cryptology – EUROCRYPT 2023, Part V*, volume 14008 of *Lecture Notes in*

Computer Science, pages 448–471, Lyon, France, April 23–27, 2023. Springer, Cham, Switzerland. 213

[Moo20] Nick Mooney. An Introduction to the Noise Protocol Framework, March 2020. https://duo.com/labs/tech-notes/noise-protocol-framework-intro. 157

[MS20] Michele Mosca and Douglas Stebila. Open quantum safe, 2020. https://openquantumsafe.org/ (accessed 2020-01-30). 110

[MSCB13] Simon Meier, Benedikt Schmidt, Cas Cremers, and David Basin. The TAMARIN prover for the symbolic analysis of security protocols. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, volume 8044 of *LNCS*, pages 696–701. Springer, 2013. http://www-oldurls.inf.ethz.ch/personal/basin/pubs/cav13.pdf. 92

[NAB+20] Michael Naehrig, Erdem Alkim, Joppe Bos, Léo Ducas, Karen Easterbrook, Brian LaMacchia, Patrick Longa, Ilya Mironov, Valeria Nikolaenko, Christopher Peikert, Ananth Raghunathan, and Douglas Stebila. FrodoKEM. Technical report, National Institute of Standards and Technology, 2020. available at https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions. 2, 212

[NL18] Yoav Nir and Adam Langley. ChaCha20 and Poly1305 for IETF protocols. IETF RFC 8439, 2018. https://tools.ietf.org/pdf/rfc8200.pdf. 87

[OBR+20] Emad Omara, Benjamin Beurdouche, Eric Rescorla, Srinivas Inguva, Albert Kwon, and Alan Duric. The Messaging Layer Security (MLS) Architecture. Internet-Draft draft-ietf-mls-architecture-05, Internet Engineering Task Force, July 2020. Work in Progress. 29

[Op22] OpenSSH team. Openssh 9.0 release note, 2022. https://www.openssh.com/txt/release-9.0. 213

[OWK23] Mike Ounsworth, Aron Wussler, and Stavros Kousidis. Combiner function for hybrid key encapsulation mechanisms (Hybrid KEMs). Internet-Draft draft-ounsworth-cfrg-kem-combiners-04, Internet Engineering Task Force, July 2023. Work in Progress. 221

[PC18] Trevor Perrin and Justin Cormack. Static-Static Pattern Modifiers for Noise, 2018. Revision 1, 2018-11-18, unofficial/unstable, https://github.com/noiseprotocol/noise_ss_spec. 171

[Pei20] Chris Peikert. He gives C-sieves on the CSIDH. In Anne Canteaut and Yuval Ishai, editors, *Advances in Cryptology – EUROCRYPT 2020, Part II*, volume 12106 of *Lecture Notes in Computer Science*, pages 463–492, Zagreb, Croatia, May 10–14, 2020. Springer, Cham, Switzerland. 89, 159

[Per] Trevor Perrin. Noise protocol framework. https://noiseprotocol.org/noise.pdf (Revision 34 vom 2018-07-11). 87, 91, 157, 158

[Per17] Trevor Perrin. The Noise Protocol Framework, December 2017. https://media.ccc.de/v/34c3-9222-the_noise_protocol_framework. 157

[Per18] Trevor Perrin. KEM-based hybrid forward secrecy for Noise, 2018. https://github.com/noiseprotocol/noise_hfs_spec/blob/master/output/noise_hfs.pdf. 91

[PFH+22] Thomas Prest, Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. FALCON. Technical report, National Institute of Standards and Technology, 2022. available at https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022. 2, 215

[PST20] Christian Paquin, Douglas Stebila, and Goutam Tamvada. Benchmarking post-quantum cryptography in TLS. In Jintai Ding and Jean-Pierre Tillich, editors, *Post-Quantum Cryptography - 11th International Conference, PQCrypto 2020*, pages 72–91, Paris, France, April 15–17, 2020. Springer, Cham, Switzerland. 201, 222

[RD12] Juliano Rizzo and Thai Duong. CVE-2012-4929 "CRIME". Available from MITRE, CVE-2012-4929., September 2012. 3

[RMS18] Paul Rösler, Christian Mainka, and Jörg Schwenk. More is less: On the end-to-end security of group chats in signal, WhatsApp, and threema. In *2018 IEEE European Symposium on Security and Privacy*, pages 415–429, London, United Kingdom, April 24–26, 2018. IEEE Computer Society Press. 29, 31, 33, 35, 40

[Rob] Raphael Robert. Re: [mls] deniability without pairwise channels. on the MLS mailing-list. 30

[Rob23] Damien Robert. Breaking SIDH in polynomial time. In Carmit Hazay and Martijn Stam, editors, *Advances in Cryptology – EUROCRYPT 2023, Part V*, volume 14008 of *Lecture Notes in Computer Science*, pages 472–503, Lyon, France, April 23–27, 2023. Springer, Cham, Switzerland. 213

[Rog02] Phillip Rogaway. Authenticated-encryption with associated-data. In Vijayalakshmi Atluri, editor, *ACM CCS 2002: 9th Conference on Computer and Communications Security*, pages 98–107, Washington, DC, USA, November 18–22, 2002. ACM Press. 17, 177, 275

[RS06] Alexander Rostovtsev and Anton Stolbunov. Public-Key Cryptosystem Based On Isogenies. Cryptology ePrint Archive, Report 2006/145, 2006. 10

[RSW96] Ronald L Rivest, Adi Shamir, and David A Wagner. Time-lock puzzles and timed-release crypto. 1996. 60, 78

[SAB+22] Peter Schwabe, Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Gregor Seiler, Damien Stehlé, and Jintai Ding. CRYSTALS-KYBER. Technical report, National Institute of Standards and Technology, 2022. available at https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022. 2, 211

[Sch16] Berry Schoenmakers. Explicit optimal binary pebbling for one-way hash chain reversal. In Jens Grossklags and Bart Preneel, editors, *FC 2016: 20th International Conference on Financial Cryptography and Data Security*, volume 9603 of *Lecture Notes in Computer Science*, pages 299–320, Christ Church, Barbados, February 22–26, 2016. Springer Berlin Heidelberg, Germany. 60

[SH19] Michael Schliep and Nicholas Hopper. End-to-end secure mobile group messaging with conversation integrity and deniability. In *18th ACM Workshop on Privacy in the Electronic Society*, WPES'19, page 55–73, New York, NY, USA, 2019. Association for Computing Machinery. 28

[Sho94] Peter W. Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *35th Annual Symposium on Foundations*

of Computer Science, pages 124–134, Santa Fe, NM, USA, November 20–22, 1994. IEEE Computer Society Press. 1

[SPG19] Michael Specter, Sunoo Park, and Matthew Green. KeyForge: Mitigating email breaches with forward-forgeable signatures. Cryptology ePrint Archive, Report 2019/390, 2019. 30

[SSW20] Peter Schwabe, Douglas Stebila, and Thom Wiggers. Post-quantum TLS without handshake signatures. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020: 27th Conference on Computer and Communications Security*, pages 1461–1480, Virtual Event, USA, November 9–13, 2020. ACM Press. 201, 222, 224

[SVH18] Michael Schliep, Eugene Vasserman, and Nicholas Hopper. Consistent synchronous group off-the-record messaging with sym-gotr. *PoPETs*, 2018(3):181 – 202, 01 Jun. 2018. 28

[SWZ16] John M. Schanck, William Whyte, and Zhenfei Zhang. Quantum-Safe Hybrid (QSH) Ciphersuite for Transport Layer Security (TLS) version 1.2. Internet-Draft draft-whyte-qsh-tls12-02, Internet Engineering Task Force, July 2016. Work in Progress. 222

[tMp] the MLS-project. Issue 50: Provide details about deniability. 28

[Tor18] Linus Torvalds. Re: [GIT] Networking. Posting to the Linux kernel mailing list, 2018. `http://lkml.iu.edu/hypermail/linux/kernel/1808.0/02472.html`. 88

[tOt] the OTRv4 team. Off-the-record messaging protocol version 4 (draft). available at `https://bugs.otr.im/otrv4/otrv4`, commit `127793d9` from 28. 10. 2019. 28, 32

[TPD21] Chengdong Tao, Albrecht Petzoldt, and Jintai Ding. Efficient key recovery for all HFE signature variants. In Tal Malkin and Chris Peikert, editors, *Advances in Cryptology – CRYPTO 2021, Part I*, volume 12825 of *Lecture Notes in Computer Science*, pages 70–93, Virtual Event, August 16–20, 2021. Springer, Cham, Switzerland. 218

[UDB+15a] N. Unger, S. Dechand, J. Bonneau, S. Fahl, H. Perl, I. Goldberg, and M. Smith. Sok: Secure messaging. In *2015 IEEE Symposium on Security and Privacy*, pages 232–249, 2015. 28

[UDB+15b] Nik Unger, Sergej Dechand, Joseph Bonneau, Sascha Fahl, Henning Perl, Ian Goldberg, and Matthew Smith. SoK: Secure messaging. In *2015 IEEE Symposium on Security and Privacy*, pages 232–249, San Jose, CA, USA, May 17–21, 2015. IEEE Computer Society Press. 31, 33, 42

[UG18] Nik Unger and Ian Goldberg. Improved strongly deniable authenticated key exchanges for secure messaging. *Proceedings on Privacy Enhancing Technologies*, 2018(1):21–66, January 2018. 29

[Val21] Filippo Valsorda. Twitter-Survey on Crypto-Agility, April 2021. https://twitter.com/FiloSottile/status/1386751406758105089. 161

[VP17] Mathy Vanhoef and Frank Piessens. Key reinstallation attacks: Forcing nonce reuse in WPA2. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017: 24th Conference on Computer and Communications Security*, pages 1313–1328, Dallas, TX, USA, October 31 – November 2, 2017. ACM Press. 3

[XLL+18] Haiyang Xue, Xianhui Lu, Bao Li, Bei Liang, and Jingnan He. Understanding and constructing ake via double-key key encapsulation mechanism. In Thomas Peyrin and Steven Galbraith, editors, *Advances in Cryptology – ASIACRYPT 2018*, volume 11274 of *LNCS*, pages 158–189. Springer, 2018. https://eprint.iacr.org/2018/817. 91

[XXW+19] Xiu Xu, Haiyang Xue, Kunpeng Wang, , Man Ho Au, Bei Liang, and Song Tian. Strongly secure authenticated key exchange from supersingular isogenies. In Steven D. Galbraith and Shiho Moriai, editors, *Advances in Cryptology – ASIACRYPT 2019*, volume 11921 of *LNCS*, pages 278–308. Springer, 2019. https://eprint.iacr.org/2018/760. 92

[YMT17] Yusuke Yoshida, Kirill Morozov, and Keisuke Tanaka. CCA2 key-privacy for code-based encryption in the standard model. In Tanja Lange and Tsuyoshi Takagi, editors, *Post-Quantum Cryptography - 8th International Workshop, PQCrypto 2017*, pages 35–50, Utrecht, The Netherlands, June 26–28, 2017. Springer, Cham, Switzerland. 107

[ZCD+20] Greg Zaverucha, Melissa Chase, David Derler, Steven Goldfeder, Claudio Orlandi, Sebastian Ramacher, Christian Rechberger, Daniel Slamanig, Jonathan Katz, Xiao Wang, Vladmir Kolesnikov, and Daniel Kales. Picnic. Technical report, National Institute of Standards and Technology, 2020. available at https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions. 218

[ZZD+15] Jiang Zhang, Zhenfeng Zhang, Jintai Ding, Michael Snook, and Özgür Dagdelen. Authenticated key exchange from ideal lattices. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology – EUROCRYPT 2015*, volume 9057 of *LNCS*, pages 719–751. Springer, 2015. https://eprint.iacr.org/2014/589/. 92

Summary

Cryptographic Protocols in a Post-Quantum World

This thesis about cryptographic protocols consists of several parts:

Firstly an introduction of epochal signatures, which are a primitive that can provide authenticity like regular signatures, but does so in a way that quickly becomes deniable. The originally intended application for them are group-chats, for which we introduced a new formal deniability definition (to our knowledge the first one in this setting) and proved that a large class of protocols that use signatures for authenticity (including Message Layer Security, "MLS") can be made deniable by substituting those signatures with epochal signatures.

Secondly we introduced PQWireGuard, a version of the WireGuard-handshake that only uses primitives that can withstand attacks with quantum computers and proved that the converted version satisfies the same security-notion that a previous analysis of the (pre-quantum) WireGuard-handshake had demonstrated for it.

Thirdly we applied the experience gathered during the PQWireGuard-project to introduce post-quantum security to "Noise", a widely used framework for secure-channels. Our analysis shows that PQNoise provides alternatives that meet or exceed those which have been shown or conjectured in previous work for classical Noise patterns.

Lastly we introduce a key-update mechanism for use in satellite communication, including a discussion of the real-world trade-offs of different instantiations to primitives.

Curriculum Vitae

Fiona Johanna Weber was born in February 1992 in Garmisch-Partenkirchen, a town in the deep south of Germany. She grew up in Massenbachhausen, a village in the south-west of the country and went to the Robert-Mayer-Gymnasium in the nearby city of Heilbronn.

She then went to Karlsruhe to initially study applied biology at KIT, before switching to computer science after her first year there. She finished her bachelor's degree in 2016 with a thesis on "Semantische Objektorientierte Programmierung" and her master's degree in March 2019 with the thesis "Privacy Preserving Advertisements".

After the completion of her master's, she moved to Eindhoven in 2019 to to pursue a Ph.D. at Eindhoven University of Technology in the Netherlands. Fiona's position as a Ph.D. Teaching Assistant gave her the room to research cryptographic protocols with a particular focus on key-establishment and the opportunity to spend more time teaching students in cryptography, than it would regularly have been the case. This dissertation stands as the result of the research undertaken throughout her Ph.D., which was supervised by Andreas Hülsing.