Post Quantum Noise

Yawning Angel* Benjamin Dowling† Andreas Hülsing‡ Peter Schwabe§
Fiona Johanna Weber¶

September 25, 2023

Abstract

We introduce PQNoise, a post-quantum variant of the Noise framework. We demonstrate that it is possible to replace the Diffie-Hellman key-exchanges in Noise with KEMs in a secure way. A challenge is the inability to combine key pairs of KEMs, which can be resolved by certain forms of randomness-hardening for which we introduce a formal abstraction. We provide a generic recipe to turn classical Noise patterns into PQNoise patterns. We prove that the resulting PQNoise patterns achieve confidentiality and authenticity in the fACCE-model. Moreover we show that for those classical Noise-patterns that have been conjectured or proven secure in the fACCE-model our matching PQNoise-patterns eventually achieve the same security. Our security proof is generic and applies to any valid PQNoise pattern. This is made possible by another abstraction, called a hash-object, which hides the exact workings of how keying material is processed in an abstract stateful object that outputs pseudorandom keys under different corruption patterns. We also show that the hash chains used in Noise are a secure hash-object. Finally, we demonstrate the practicality of PQNoise delivering benchmarks for several base patterns.

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 initator'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 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

*Oasis Labs, Email: yawning@oasislabs.com
†University of Sheffield, Email: b.dowling@sheffield.ac.uk
‡TU Eindhoven, E-mail: andreas@huelsing.net
§MPI-SP, E-mail: peter@cryptojedi.org
¶TU Eindhoven, E-mail: crypto@fionajw.de
Author list in alphabetical order, see: `https://www.ams.org/profession/leaders/culture/CultureStatement04.pdf`

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].

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 paper 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 amount 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 exists [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 roundtrips, 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 chose 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 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 to simplify the freshness conditions significantly.

Finally we present a proof-of-concept implementation of PQNoise in Go and report on benchmarking results. The results 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 later is a property whose desirability is getting called increasingly into question as more and more newer protocols don't offer which matches community-surveys [Val21]. The software is available from:
https://gitlab.com/yawning/nyquist/-/tree/experimental/pqnoise.

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.

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 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 preservers confidentiality if there is no additional corruption.) As depicted in Algorithm 1, during Send the sender encapsulates a key $k_{\mathcal{X}}$ to the receiver's public key $pk_{\mathcal{X}}$ using hardened randomness (see Section 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 ciphertext $ct_{\mathcal{X}}$ (together with possible further payload pl that doesn't further affect the KEM-operation, see Appendix G) 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 keychain 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 1: Transmission of KEM-ciphertexts.

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 Ginstead. 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 Gand the compiler at `https://florianjw.de/diverses/pqnoise-codegen.tar.bz2`.

To give an illustrative example of how PQNoise and Noise differ we refer to Figure 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.

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 academic 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 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 concrete 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 GenRand. GenKey is a probabilistic algorithm that returns a long-term key sk. GenRand then takes sk and some random coins and returns a pseudo-random value r, where r is indistinguishable from a true random value if either sk is uncorrupted and the random-coins fresh (but possibly known to the adversary) or if the random-coins are uncorrupted. Additionally we allow but do not require 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. We provide a more formal definition in

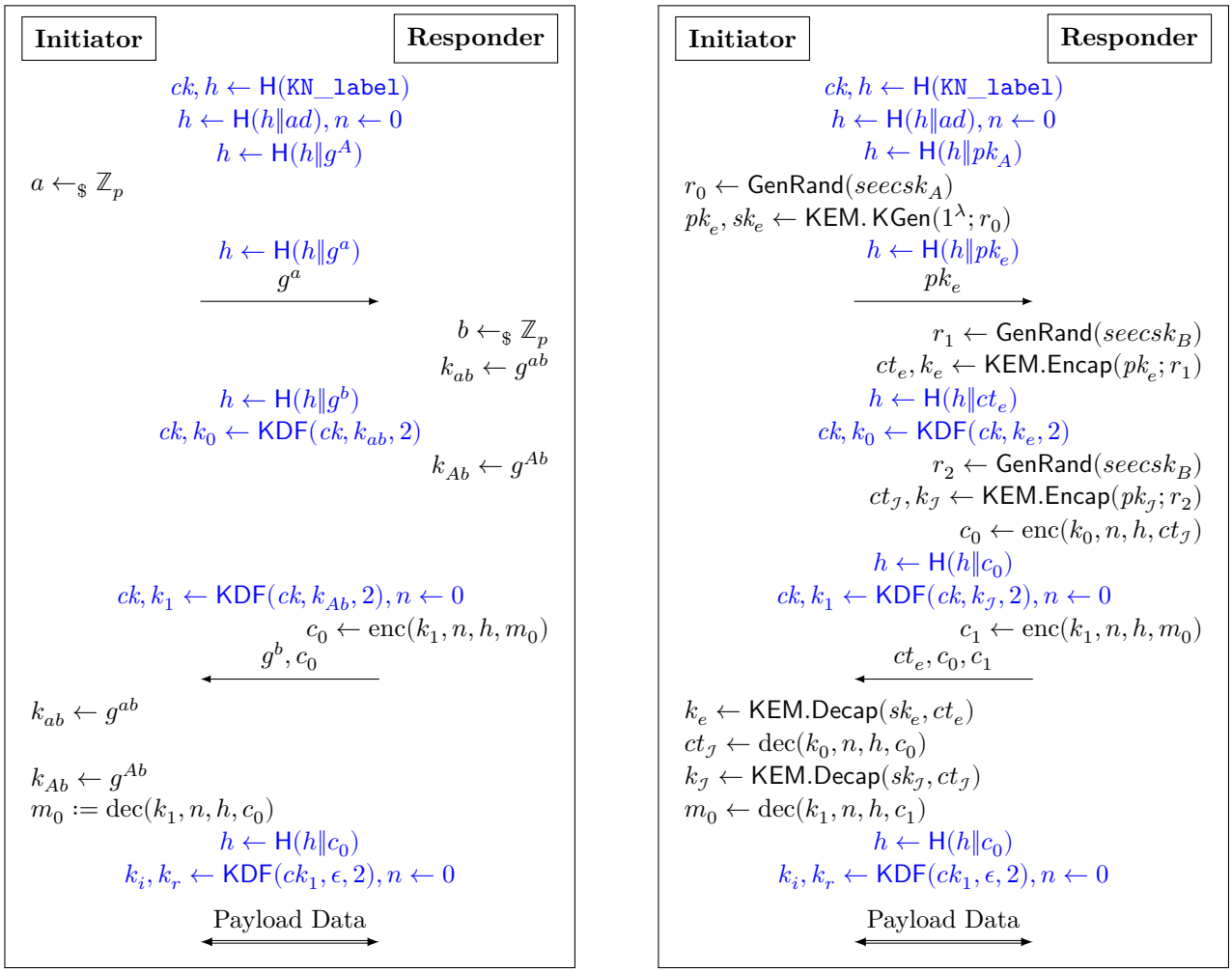

Figure 1: The KN patterns of classical Noise (left) and PQNoise (right). For reasons of space we use the following conventions here: highlighted actions are performed by both parties as soon as they receive all necessary values to perform the computations in question. If any decryption- or decapsulation-algorithm returns $\perp$, the party in question aborts the protocol.

applabel Cand "PRP-SEEC" as a simple, exemplary instantiation in Appendix E .

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 uncorrupted), 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 uncorrupted 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 excep-

tion 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 initiators 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 parties 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.

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, put to also do that as efficient 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` always precedes `skem` which 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 2 and provide more detailed descriptions and a comparison with their Noise counterparts in the Appendices G and H.

```
pqNN:          pqNK:          pqNX:
-> e           <- s           -> e
<- ekem        ...            <- ekem, s
               -> skem, e     -> skem
               <- ekem

pqKN:          pqKK:          pqKX:
-> s           -> s           -> s
...            <- s           ...
-> e           ...            -> e
<- ekem, skem  -> skem, e     <- ekem, skem, s
               <- ekem, skem  -> skem

pqXN:          pqXK:          pqXX:
-> e           <- s           -> e
<- ekem        ...            <- ekem, s
-> s           -> skem, e     -> skem, s
<- skem        <- ekem        <- skem
               -> s
               <- skem

pqIN:          pqIK:          pqIX:
-> e, s        <- s           -> e, s
<- ekem, skem  ...            <- ekem, skem, s
               -> skem, e, s  -> skem
               <- ekem, skem
```

Figure 2: The interactive fundamental PQNoise patterns.

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. $\mathtt{au}^{\rho}$ 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* corrupted 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 D.

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.

Definition 1 (Flexible ACCE)**.** A flexible ACCE protocol fACCE is a tuple of four algorithms 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:

KGen $\rightarrow_{\$} (sk, pk)$ generates long-term keys where $sk \in \mathcal{LSK}, pk \in \mathcal{LPK}$. 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).

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 $sk \in \mathcal{LSK} \cup \{\perp\}, ppk \in \mathcal{LPK} \cup \{\perp\}, ad \in \{0,1\}^*, st \in \mathcal{ST}$.

Send$(sk, st, m) \rightarrow_{\$} (st', c)$ continues the protocol execution in a session and takes message m to output 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].

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 D, Definition 17. 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 D.

Honest partnering is defined over the transcript sent between two sessions. Intuitively, a session has an honest partner if all ciphertexts the honest partner 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 18 in Appendix D.

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 1 (see Appendix Hfor the full protocol). 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 uncom-

[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

Table 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

promised, 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.

Adversarial Model. In order to model active attacks in our environment, the security experiment provides the $\mathsf{OInit}, \mathsf{OSend}, \mathsf{ORecv}$ 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 D.

$\mathsf{OInit}(i, pk_j, \rho, ad)$ initializes a new session π_i^s (if not yet initialized) of party i to be partnered with party j, invoking
$\text{fACCE}.\mathsf{Init}(sk_i, pk_j, \rho, ad) \rightarrow_{[\pi_i^s.rand]} \pi_i^s.st$ 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 D. A flexible ACCE protocol fACCE is post-quantum secure if it is correct and $\mathsf{Adv}_{\mathcal{Q}}^{\text{fACCE}}$ is negligible for all quantum algorithms $\mathcal{Q}$ running in polynomial-time.

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.

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 2 (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 $m \in \{0,1\}^*$ and returns a new state s' and a list $h \in (\{0,1\}^\lambda)^n$ 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 (i.e. $h := s.\mathsf{input}(m)$ instead of $s, h := \mathsf{input}(s, m)$).

Definition 3 (Pseudorandom Hash-Object)**.** We say that a hash-object HO is a pseudorandom hash-object if and only if $\forall \mathcal{A} \in \mathrm{QPT}, \lambda \in \mathbb{N}$:

$$\left| \begin{array}{l} \Pr\left[\mathsf{Exp}^{PRHO}_{\mathsf{HO},\mathcal{A},0}(1^\lambda) = 1\right] \\ -\Pr\left[\mathsf{Exp}^{PRHO}_{\mathsf{HO},\mathcal{A},1}(1^\lambda) = 1\right] \end{array} \right| =: \mathsf{Adv}^{PRHO}_{\mathsf{HO},\mathcal{A}}(1^\lambda) \leq$$

$\mathrm{negl}(\lambda)$ where $\mathsf{Exp}^{PRHO}_{\mathsf{HO},\mathcal{A}}$ is defined as in Experiment 1.

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 2. 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 1. *A Noise Hash Object NHO is a secure pseudo-random Hash-Object if* $\mathsf{HMAC\text{-}HASH}$ *is a dual-prf with:* $\mathsf{Adv}^{PRHO}_{NHO,\mathcal{A},q_i}(1^\lambda) \leq$

$$\left(\begin{array}{l} \mathsf{Adv}^{CollRes}_{\mathsf{HMAC\text{-}HASH},\mathcal{A}'}(1^\lambda) + \\ \mathsf{Adv}^{PRF\text{-}SWAP}_{\mathsf{HMAC\text{-}HASH},\mathcal{A}'}(1^\lambda) + \\ (2 \cdot q) \cdot \mathsf{Adv}^{PRF}_{\mathsf{HMAC\text{-}HASH},\mathcal{A}'}(1^\lambda) \end{array} \right)$$

where q refers to the total number of oracle-queries.

We refer to Appendix Afor a proof. Intuitively the collision-resistance of HMAC-HASH implies that only identical histories result in equal states and the HMAC-HASH being a dual-PRF (see Appendix B.2) ensures that once r has been added to a chain, its first state becomes pseudorandom which is retained upon subsequent calls.

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 Appendix B). 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 $\mathcal{P}$.

Experiment 1: $\mathsf{Exp}^{PRHO}_{\mathsf{HO},\mathcal{A},b}$, the pseudo-randomness experiment for a hash-object HO.

$r \xleftarrow{\$} \{0,1\}^\lambda$
$hashes := [],\ history := [],\ j := 0$
$randomized := [0,\dots,0]$
$finalized := [0,\dots,0]$
$queries := \emptyset$
$cache := \mathsf{dict}()$
Oracle Create:
  $i, j := j, j+1$
  $hashes[i] := \mathsf{create}(1^\lambda)$
  $history[i] := []$
  return i
Oracle Rand$(i, finalize)$:
  $randomized[i] := 1$
  if *finalize*:
    return $\mathsf{Fin}(i, r)$
  else:
    return $\mathsf{In}(i, r)$
Oracle $\mathsf{In}(i, m)$:
  abort_if($finalized[i]$ $\vee$ $(history[i]\,||(\text{“Fin”}, m)) \in queries$)
  $history[i].\mathsf{append}(\text{“In”}, m)$
  $queries \cup = history[i]$
  if *randomized [i]*:
    if $cache[history[i]] \neq \perp$:
      return $cache[history[i]]$
    $h_0 := hashes[i].\mathsf{input}(m)$
    $h_1 \xleftarrow{\$} (\{0,1\}^\lambda)^n$
    $cache[history[i]] := h_b$
    return h_b
  else:
    return $hashes[i].\mathsf{input}(m)$
Oracle $\mathsf{Fin}(i, m)$:
  abort_if($finalized[i]$ $\vee$ $(history[i]\,||(\text{“In”}, m)) \in queries$)
  $finalized[i] := 1$
  $history[i].\mathsf{append}(\text{“Fin”}, m)$
  $queries \cup = history[i]$
  if *randomized [i]*:
    if $cache[history[i]] \neq \perp$:
      return $cache[history[i]]$
    $h_0 := hashes[i].\mathsf{finalize}(m)$
    $h_1 \xleftarrow{\$} (\{0,1\}^\lambda)^n$
    $cache[history[i]] := h_b$
    return h_b
  else:
    return $hashes[i].\mathsf{finalize}(m)$
return $\mathcal{A}^{\mathsf{Create},\mathsf{In},\mathsf{Fin},\mathsf{Rand}}(1^\lambda)$

Algorithm 2: Noise-compatible instantiation for the pseudo-random hash-object.

Function $\mathsf{create}(1^\lambda)$:
  return “”
Function $\mathsf{finalize}(state, m)$:
  $(h_0, [h_1.\dots, h_n]) := \mathsf{input}(state, m)$
  return $[h_0.\dots, h_{n-1}])$
Function $\mathsf{input}(state, m)$:
  $tmp := \mathsf{HMAC\text{-}HASH}(state, m)$
  $last := $ “”
  for $i \in \{0,\dots,n\}$:
    $h_i := \mathsf{HMAC\text{-}HASH}(tmp, last||\,\mathsf{byte}(i))$
    $last := h_i$
  return $h_0, [h_1.\dots, h_n]$

Theorem 2. Π *achieves initiator-authenticity in stage* $\#I+1$ *if the initiator's static key and either of the responders keys have not been corrupted with:*

$$\mathsf{Adv}^{fACCE.auth_\jmath}_{\Pi,\mathcal{A}}(1^\lambda) \leq \mathsf{Adv}^{\mathsf{CollRes}}_{\mathsf{H},\mathcal{A}'}(1^\lambda) + \tfrac{4(n_P\cdot n_S)^2}{2^\lambda} + n_P^2 \cdot n_S \cdot (\mathsf{Adv}^{SEEC}_{\Sigma,\mathcal{A}'}(1^\lambda) + \mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{IKEM,\mathcal{A}',n_S}(1^\lambda) + \mathsf{Adv}^{\mathsf{PRHO}}_{H,\mathcal{A}'}(1^\lambda) + n_K \cdot \mathsf{Adv}^{\mathsf{auth}}_{AEAD,\mathcal{A}'}(1^\lambda))$$

Theorem 3. Π' *achieves initiator-authenticity in stage* $\#I+1$ *if the initiator's static key and the responders ephemeral key have not been corrupted with:*

$$\mathsf{Adv}^{fACCE.auth_\jmath}_{\Pi,\mathcal{A}}(1^\lambda) \leq \mathsf{Adv}^{\mathsf{CollRes}}_{\mathsf{H},\mathcal{A}'}(1^\lambda) + \tfrac{4(n_P\cdot n_S)^2}{2^\lambda} + n_P^2 \cdot n_S \cdot (\mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{IKEM,\mathcal{A}',n_S}(1^\lambda) + \mathsf{Adv}^{\mathsf{PRHO}}_{H,\mathcal{A}'}(1^\lambda) + n_K \cdot \mathsf{Adv}^{\mathsf{auth}}_{AEAD,\mathcal{A}'}(1^\lambda))$$

Theorem 4. Π *achieves responder-authenticity in stage* $\#R+1$ *if the responders static key and either of the initiator's keys have not been corrupted with:*

$$\mathsf{Adv}^{fACCE.auth_\mathcal{R}}_{\Pi,\mathcal{A}}(1^\lambda) \leq \mathsf{Adv}^{\mathsf{CollRes}}_{\mathsf{H},\mathcal{A}'}(1^\lambda) + \tfrac{4(n_P\cdot n_S)^2}{2^\lambda} + n_P^2 \cdot n_S \cdot (\mathsf{Adv}^{SEEC}_{\Sigma,\mathcal{A}'}(1^\lambda) + \mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{RKEM,\mathcal{A}',n_S}(1^\lambda) + \mathsf{Adv}^{\mathsf{PRHO}}_{H,\mathcal{A}'}(1^\lambda) + n_K \cdot \mathsf{Adv}^{\mathsf{auth}}_{AEAD,\mathcal{A}'}(1^\lambda))$$

Theorem 5. Π' *achieves responder-authenticity in stage* $\#R+1$ *if the responders static key and the initiator's ephemeral key have not been corrupted with:*

$$\mathsf{Adv}^{fACCE.auth_\mathcal{R}}_{\Pi,\mathcal{A}}(1^\lambda) \leq \mathsf{Adv}^{\mathsf{CollRes}}_{\mathsf{H},\mathcal{A}'}(1^\lambda) + \tfrac{4(n_P\cdot n_S)^2}{2^\lambda} + n_P^2 \cdot n_S \cdot (\mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{RKEM,\mathcal{A}',n_S}(1^\lambda) + \mathsf{Adv}^{\mathsf{PRHO}}_{H,\mathcal{A}'}(1^\lambda) + n_K \cdot \mathsf{Adv}^{\mathsf{auth}}_{AEAD,\mathcal{A}'}(1^\lambda))$$

We provide a detailed proof in Appendix A. 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.

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 6. Π *achieves confidentiality in stage* $\#I$ *if either of the responders and the static key of the initiator is uncorrupted and the responder is an honest party with:*

$$\mathsf{Adv}^{fACCE.conf_\jmath}_{\Pi,\mathcal{A}}(1^\lambda) \leq \tfrac{4(n_P\cdot n_S)^2}{2^\lambda} + \mathsf{Adv}^{\mathsf{CollRes}}_{\mathsf{H},\mathcal{A}'}(1^\lambda) + n_P^2\cdot n_S^2 \cdot$$

$(\mathsf{Adv}^{SEEC}_{\Sigma,\mathcal{A}'}(1^\lambda) + \mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{IKEM,\mathcal{A}',n_S}(1^\lambda) + \mathsf{Adv}^{\mathsf{PRHO}}_{H,\mathcal{A}'}(1^\lambda) + (n_S-1)\cdot \mathsf{Adv}^{\mathsf{auth}}_{AEAD,\mathcal{A}'}(1^\lambda) + n_K \cdot \mathsf{Adv}^{\mathsf{IND\$\text{-}CPA}}_{AEAD,\mathcal{A}'}(1^\lambda))$

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 7. Π' *achieves confidentiality in stage #I if the responders ephemeral key and the static key of the initiator are uncorrupted and the responder is an honest party with:* $\mathsf{Adv}^{fACCE.conf_I}_{\Pi,\mathcal{A}}(1^\lambda) \leq \frac{4(n_P\cdot n_S)^2}{2^\lambda} + \mathsf{Adv}^{\mathsf{CollRes}}_{\mathsf{H},\mathcal{A}'}(1^\lambda) + n_P^2 \cdot n_S^2 \cdot (\mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{IKEM,\mathcal{A}',n_S}(1^\lambda) + \mathsf{Adv}^{\mathsf{PRHO}}_{H,\mathcal{A}'}(1^\lambda) + (n_S-1)\cdot \mathsf{Adv}^{\mathsf{auth}}_{AEAD,\mathcal{A}'}(1^\lambda) + n_K \cdot \mathsf{Adv}^{\mathsf{IND\$\text{-}CPA}}_{AEAD,\mathcal{A}'}(1^\lambda))$

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 8. Π *achieves confidentiality in stage #R if either of the initiators keys and the static key of the responder is uncorrupted and the initiator is an honest party with:* $\mathsf{Adv}^{fACCE.conf_R}_{\Pi,\mathcal{A}}(1^\lambda) \leq \frac{4(n_P\cdot n_S)^2}{2^\lambda} + \mathsf{Adv}^{\mathsf{CollRes}}_{\mathsf{H},\mathcal{A}'}(1^\lambda) + n_P^2 \cdot n_S^2 \cdot (\mathsf{Adv}^{SEEC}_{\Sigma,\mathcal{A}'}(1^\lambda) + \mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{RKEM,\mathcal{A}',n_S}(1^\lambda) + \mathsf{Adv}^{\mathsf{PRHO}}_{H,\mathcal{A}'}(1^\lambda) + (n_S-1)\cdot \mathsf{Adv}^{\mathsf{auth}}_{AEAD,\mathcal{A}'}(1^\lambda) + n_K \cdot \mathsf{Adv}^{\mathsf{IND\$\text{-}CPA}}_{AEAD,\mathcal{A}'}(1^\lambda))$

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 9. Π' *achieves confidentiality in stage #R if the initiators ephemeral key and the static key of the responder are uncorrupted and the initiator is an honest party with:* $\mathsf{Adv}^{fACCE.conf_R}_{\Pi,\mathcal{A}}(1^\lambda) \leq \frac{4(n_P\cdot n_S)^2}{2^\lambda} + \mathsf{Adv}^{\mathsf{CollRes}}_{\mathsf{H},\mathcal{A}'}(1^\lambda) + n_P^2 \cdot n_S^2 \cdot (\mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{RKEM,\mathcal{A}',n_S}(1^\lambda) + \frac{4(n_P\cdot n_S)^2}{2^\lambda} + \mathsf{Adv}^{\mathsf{PRHO}}_{H,\mathcal{A}'}(1^\lambda) + (n_S-1)\cdot \mathsf{Adv}^{\mathsf{auth}}_{AEAD,\mathcal{A}'}(1^\lambda) + n_K \cdot \mathsf{Adv}^{\mathsf{IND\$\text{-}CPA}}_{AEAD,\mathcal{A}'}(1^\lambda))$

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. We provide them in Appendix A.

The last two of the six confidentiality theorems deal with uncorrupted ephemeral keys and are largely analogous to the previous four besides that.

Theorem 10. 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:*

$\mathsf{Adv}^{fACCE.conf_E}_{\Pi,\mathcal{A}}(1^\lambda) \leq \frac{4(n_P\cdot n_S)^2}{2^\lambda} + \mathsf{Adv}^{\mathsf{CollRes}}_{\mathsf{H},\mathcal{A}'}(1^\lambda) + n_P^2 \cdot n_S^2 \cdot (2\cdot \mathsf{Adv}^{SEEC}_{\Sigma,\mathcal{A}'}(1^\lambda) + \mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{EKEM,\mathcal{A}',1}(1^\lambda) + \mathsf{Adv}^{\mathsf{PRHO}}_{H,\mathcal{A}'}(1^\lambda) + n_K \cdot \mathsf{Adv}^{\mathsf{IND\$\text{-}CPA}}_{AEAD,\mathcal{A}'}(1^\lambda))$

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 11. Π' *achieves confidentiality in stage #E if neither the initiators nor the responders ephemeral keys are uncorrupted and both are honest partners with:*

$\mathsf{Adv}^{fACCE.conf_E}_{\Pi,\mathcal{A}}(1^\lambda) \leq \frac{4(n_P\cdot n_S)^2}{2^\lambda} + \mathsf{Adv}^{\mathsf{CollRes}}_{\mathsf{H},\mathcal{A}'}(1^\lambda) + n_P^2 \cdot n_S^2 \cdot (\mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{EKEM,\mathcal{A}',1}(1^\lambda) + \mathsf{Adv}^{\mathsf{PRHO}}_{H,\mathcal{A}'}(1^\lambda) + n_K \cdot \mathsf{Adv}^{\mathsf{IND\$\text{-}CPA}}_{AEAD,\mathcal{A}'}(1^\lambda))$

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). We provide them in Appendix A.

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 2. When compared with the partially conjectured results for Noise [DRS20] (see Appendix F) we see that PQNoise ends up with the same security as classical Noise eventually.

5. Implementation

We implement PQNoise as an extension of the ``nyquist'' implementation of Noise by Angel in the Go programming language [Ang]. As underlying instantiation of all KEMs we use 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, but also SIKE [ACC+20], it would be easy to change to a different instantiation of the KEMs.

When comparing performance between PQNoise handshakes and corresponding Noise handshakes, we notice that *computationally*, i.e., in terms of CPU cycles, PQNoise is 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.

Table 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

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 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, we consider 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 amount of round trips is the same, while for the `pqXX` pattern an additional message from the responder is required compared to `XX`. We run benchmarks on a machine with two Intel Xeon Gold 6230 CPUs and 196 GB of RAM. The experimental setup is using the Linux kernel's network-emulation features and is largely following the setup used in [PST20] and [SSW20]. For each of the patterns we take 1000 measurements of the time it takes to perform a handshake, independently for initiator and responder. We perform this measurements 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 3. 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.

Table 3: 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

6. Acknowledgement

We thank Trevor Perrin and Denisa Greconici for many helpful discussions.

This work has been supported by the Dutch Research Council (NWO) through VIDI grant No. VI.Vidi.193.066, by the European Research Council through Starting Grant No. 805031 (EPOQUE), and by Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) as part of the Excellence Strategy of the German Federal and State Governments -- EXC 2092 CASA - 390781972.

References

[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. 12

[ACC+20] Reza Azarderakhsh, Matthew Campagna, Craig Costello, Luca De Feo, Basil Hess, Aaron Hutchinson, Amir Jalali, Koray Karabina, Brian Koziel, Brian LaMacchia, Patrick Longa, Michael Naehrig, Geovandro Pereira, Joost Renes, Vladimir Soukharev, and David Urbanik. Supersingular isogeny key encapsulation. Round-3 submission to the NIST PQC project, 2020. https://sike.org/#specification. 12

[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 Computer Security Foundations Symposium*, pages 277--287. IEEE Computer Society Press, 2020. 4, 23

[Ang] Yawning Angel. nyquist - a Noise protocol framework implementation. https://github.com/Yawning/nyquist. 12

[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. 2

[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. 12

[Ber06] Daniel J. Bernstein. Curve25519: New Diffie-Hellman speed records. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *PKC 2006*, volume 3958 of *LNCS*, pages 207--228. Springer, Heidelberg, April 2006. 2

[BL] Daniel J. Bernstein and Tanja Lange. eBACS: ECRYPT benchmarking of cryptographic systems. https://bench.cr.yp.to (accessed 29 Sep 2021). 12

[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, *EUROCRYPT 2019, Part II*, volume 11477 of *LNCS*, pages 409--441. Springer, Heidelberg, May 2019. 2

[BLN16] Daniel J Bernstein, Tanja Lange, and Ruben Niederhagen. Dual ec: A standardized back door. In *The New Codebreakers*, pages 256--281. Springer, 2016. 4

[BS20] Xavier Bonnetain and André Schrottenloher. Quantum security analysis of CSIDH. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part II*, volume 12106 of *LNCS*, pages 493--522. Springer, Heidelberg, May 2020. 2

[CGS+20] Cas Cremers, Luke Garratt, Stanislav V. Smyshlyaev, Nick Sullivan, and Christopher A. Wood. Randomness Improvements for Security Protocols. RFC 8937, October 2020. 4

[CK01] Ran Canetti and Hugo Krawczyk. Analysis of key-exchange protocols and their use for building secure channels. In Birgit Pfitzmann, editor, *EUROCRYPT 2001*, volume 2045 of *LNCS*, pages 453--474. Springer, Heidelberg, May 2001. 2

[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, *ASIACRYPT 2018, Part III*, volume 11274 of *LNCS*, pages 395--427. Springer, Heidelberg, December 2018. 2

[DH76] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644--654, 1976. 1

[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. 4

[DP18] Benjamin Dowling and Kenneth G. Paterson. A cryptographic analysis of the WireGuard protocol. In Bart Preneel and Frederik Vercauteren, editors, *ACNS 18*, volume 10892 of *LNCS*, pages 3--21. Springer, Heidelberg, July 2018. 8

[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, Part I*, volume 12110 of *LNCS*, pages 341--373. Springer, Heidelberg, May 2020. 2, 4, 6, 7, 12, 21, 23, 30, 31

[FHKP12] Eduarda S.V. Freire, Dennis Hofheinz, Eike Kiltz, and Kenneth G. Paterson. Non-interactive key exchange. Cryptology ePrint Archive, Report 2012/732, 2012. https://eprint.iacr.org/2012/732. 2

[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*, volume 7293 of *LNCS*, pages 467--484. Springer, Heidelberg, May 2012. 4

[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. 2

[HNS+21] Andreas Hülsing, Kai-Chun Ning, Peter Schwabe, Florian Weber, and Philip R. Zimmermann. Post-quantum WireGuard. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 304--321. IEEE Computer Society, 2021. https://cryptojedi.org/papers/#pqwireguard. 2, 3, 8

[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. 2

[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/. 12

[KNB19] Nadim Kobeissi, Georgio Nicolas, and Karthikeyan Bhargavan. Noise explorer: Fully automated modeling and verification for arbitrary noise protocols. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 356--370. IEEE, 2019. 2

[Kra05] Hugo Krawczyk. HMQV: A high-performance secure Diffie-Hellman protocol. In Victor Shoup, editor, *CRYPTO 2005*, volume 3621 of *LNCS*, pages 546--566. Springer, Heidelberg, August 2005. 2, 4

[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*, volume 4784 of *LNCS*, pages 1--16. Springer, Heidelberg, November 2007. 2, 4, 8

[Moo20] Nick Mooney. An Introduction to the Noise Protocol Framework, March 2020. https://duo.com/labs/tech-notes/noise-protocol-framework-intro. 1

[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. 6

[Pei20] Chris Peikert. He gives C-sieves on the CSIDH. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part II*, volume 12106 of *LNCS*, pages 463--492. Springer, Heidelberg, May 2020. 2

[Per] Trevor Perrin. Noise protocol framework. https://noiseprotocol.org/noise.pdf (Revision 34 vom 2018-07-11). 1

[Per17] Trevor Perrin. The Noise Protocol Framework, December 2017. https://media.ccc.de/v/34c3-9222-the_noise_protocol_framework. 1

[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. Springer, Heidelberg, 2020. 13

[Rog02] Phillip Rogaway. Authenticated-encryption with associated-data. In Vijayalakshmi Atluri, editor, *ACM CCS 2002*, pages 98--107. ACM Press, November 2002. 9, 26

[SSW20] Peter Schwabe, Douglas Stebila, and Thom Wiggers. Post-quantum TLS without handshake signatures. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, CCS'20*, pages 1461--1480. ACM, 2020. https://cryptojedi.org/papers/#kemtls. 13

[Val21] Filippo Valsorda. Twitter-Survey on Crypto-Agility, April 2021. https://twitter.com/FiloSottile/status/1386751406758105089. 3

A. Proofs

Theorem 1. *A Noise Hash Object NHO is a secure pseudo-random Hash-Object if* HMAC-HASH *is a dual-prf with:* $\mathsf{Adv}^{PRHO}_{NHO,\mathcal{A},q_i}(1^\lambda) \leq \left(\begin{array}{l} \mathsf{Adv}^{CollRes}_{\mathsf{HMAC\text{-}HASH},\mathcal{A}'}(1^\lambda) + \\ \mathsf{Adv}^{PRF\text{-}SWAP}_{\mathsf{HMAC\text{-}HASH},\mathcal{A}'}(1^\lambda) + \\ (2 \cdot q) \cdot \mathsf{Adv}^{PRF}_{\mathsf{HMAC\text{-}HASH},\mathcal{A}'}(1^\lambda) \end{array} \right)$ *where q refers to the total number of oracle-queries.*

Proof. We use game-hopping to show the claim. Let q_{X} be the total number of calls to the X-oracle and $\Pr[\mathrm{diff}_Y]$ 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 HMAC-HASH that produce the same output. To see that this modification is undetectable we initialize a collision-resistance-challenger and use its 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 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:

$$\Pr[\text{diff}_1] \leq \mathsf{Adv}^{\mathsf{CollRes}}_{\mathsf{HMAC\text{-}HASH},\mathcal{A}'}(1^\lambda)$$

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, irrespectively 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 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-Challengers challenge-bit is zero, then it samples r randomly and computes *tmp* as $\mathsf{HMAC\text{-}HASH}(state, r)$ and we are in *Game 1*. Otherwise the values for *tmp* are sampled at random and we are in *Game 2*. Thus we find:

$$\Pr[\text{diff}_2] \leq \mathsf{Adv}^{\mathsf{PRF\text{-}SWAP}}_{\mathsf{HMAC\text{-}HASH},\mathcal{A}'}(1^\lambda)$$

In ***Game 3*** we replace the results of all invocations of 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 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 HMAC-HASH and query it with m whenever we would compute $\mathsf{HMAC\text{-}HASH}(tmp, m)$ 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 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-Challengers challenge-bit is zero, then it returns $\mathsf{HMAC\text{-}HASH}(tmp_i, m)$ 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:

$$\Pr[\text{diff}_{2.i}] \leq \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{HMAC\text{-}HASH},\mathcal{A}'}(1^\lambda)$$

Since there are at most q_{Rand} different sub-games, we can conclude by setting *Game 3* = *Game* $2.q_{\mathsf{Rand}}$ and summarizing the losses of all sub-games that:

$$\Pr[\text{diff}_3] \leq q_{\mathsf{Rand}} \cdot \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{HMAC\text{-}HASH},\mathcal{A}'}(1^\lambda)$$

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 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 input/finalize in the j'th invocation of In and Fin (=: tmp_j) with a random value, except for maintaining consistency between identical invocations of HMAC-HASH. To show that this replacement is sound we initialize a PRF-Challenger for HMAC-HASH and replace the invocation of HMAC-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-Challengers challenge-bit is zero, then the oracle returns $\mathsf{HMAC\text{-}HASH}(k, state[i])$ 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:

$$\Pr[\text{diff}_{3.j.1}] \leq \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{HMAC\text{-}HASH},\mathcal{A}'}(1^\lambda)$$

In ***Game 3.j.1*** we replace the return-values and resulting states of all invocations of input and finalize that use the value of tmp_j (as defined in the previous sub-game) by random values, such that identical invocations of HMAC-HASH do however use the same values. To show that this replacement is sound we initialize a PRF-Challenger for HMAC-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-Challengers challenge-bit is zero, then it answers all queries for a value m̆ by computing $\mathsf{HMAC\text{-}HASH}(k, m)$ 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:

$$\Pr[\text{diff}_{3.j.1}] \leq \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{HMAC\text{-}HASH},\mathcal{A}'}(1^\lambda)$$

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 1. □

Theorem 2. Π *achieves initiator-authenticity in stage* $\#I+1$ *if the initiator's static key and either of the responders keys have not been corrupted with:*

$$\mathsf{Adv}^{fACCE.auth_J}_{\Pi,\mathcal{A}}(1^\lambda) \leq \mathsf{Adv}^{\mathsf{CollRes}}_{\mathsf{H},\mathcal{A}'}(1^\lambda) + \tfrac{4(n_P \cdot n_S)^2}{2^\lambda} + n_P^2 \cdot n_S \cdot (\mathsf{Adv}^{SEEC}_{\Sigma,\mathcal{A}'}(1^\lambda) + \mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{IKEM,\mathcal{A}',n_S}(1^\lambda) + \mathsf{Adv}^{\mathsf{PRHO}}_{H,\mathcal{A}'}(1^\lambda) + n_K \cdot \mathsf{Adv}^{\mathsf{auth}}_{AEAD,\mathcal{A}'}(1^\lambda))$$

Theorem 3. Π' *achieves initiator-authenticity in stage* $\#I+1$ *if the initiator's static key and the responders ephemeral key have not been corrupted with:*

$\mathsf{Adv}^{fACCE.auth_I}_{\Pi,\mathcal{A}}(1^\lambda) \leq \mathsf{Adv}^{\mathsf{CollRes}}_{\mathsf{H},\mathcal{A}'}(1^\lambda) + \frac{4(n_P \cdot n_S)^2}{2^\lambda} + n_P^2 \cdot n_S \cdot (\mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{IKEM,\mathcal{A}',n_S}(1^\lambda) + \mathsf{Adv}^{\mathsf{PRHO}}_{H,\mathcal{A}'}(1^\lambda) + n_K \cdot \mathsf{Adv}^{\mathsf{auth}}_{AEAD,\mathcal{A}'}(1^\lambda))$

Theorem 4. Π *achieves responder-authenticity in stage* $\#R+1$ *if the responders static key and either of the initiator's keys have not been corrupted with:*

$\mathsf{Adv}^{fACCE.auth_R}_{\Pi,\mathcal{A}}(1^\lambda) \leq \mathsf{Adv}^{\mathsf{CollRes}}_{\mathsf{H},\mathcal{A}'}(1^\lambda) + \frac{4(n_P \cdot n_S)^2}{2^\lambda} + n_P^2 \cdot n_S \cdot (\mathsf{Adv}^{SEEC}_{\Sigma,\mathcal{A}'}(1^\lambda) + \mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{RKEM,\mathcal{A}',n_S}(1^\lambda) + \mathsf{Adv}^{\mathsf{PRHO}}_{H,\mathcal{A}'}(1^\lambda) + n_K \cdot \mathsf{Adv}^{\mathsf{auth}}_{AEAD,\mathcal{A}'}(1^\lambda))$

Theorem 5. Π' *achieves responder-authenticity in stage* $\#R+1$ *if the responders static key and the initiator's ephemeral key have not been corrupted with:*

$\mathsf{Adv}^{fACCE.auth_R}_{\Pi,\mathcal{A}}(1^\lambda) \leq \mathsf{Adv}^{\mathsf{CollRes}}_{\mathsf{H},\mathcal{A}'}(1^\lambda) + \frac{4(n_P \cdot n_S)^2}{2^\lambda} + n_P^2 \cdot n_S \cdot (\mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{RKEM,\mathcal{A}',n_S}(1^\lambda) + \mathsf{Adv}^{\mathsf{PRHO}}_{H,\mathcal{A}'}(1^\lambda) + n_K \cdot \mathsf{Adv}^{\mathsf{auth}}_{AEAD,\mathcal{A}'}(1^\lambda))$

Proof. We show the theorems by contraction, assuming that there is an adversary that can cause $win \leftarrow 1$ to be set. For this we use game hoping: 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 2, and in Theorem 4 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 $\mathsf{GenRand}$ 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 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 change the game and note that we are only considering cases where the randomness is uncorrupted in the first place, giving us:

$$\Pr[\text{break}_3] = \Pr[\text{break}_4]$$

In ***Game 5*** we replace the key $k_{\mathcal{C}}$ encapsulated in the ciphertext $c_{\mathcal{C}}$ under $\mathcal{C}$'s public key with a uniformly random key. To show that this replacement is sound we initialize an $\mathsf{IND\text{-}CCA}$-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:

$$\Pr[\text{break}_4] \leq \Pr[\text{break}_5] + \mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{XKEM,\mathcal{A}',n_S}(1^\lambda)$$

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 $\mathsf{input/finalize}$ functions of HO she will instead invoke the $\mathsf{In/Fin}$ 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:

$$\Pr[\text{break}_5] \leq \Pr[\text{break}_6] + \mathsf{Adv}^{PRHO}_{\mathsf{HO},\mathcal{A}'}(1^\lambda)$$

(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:

$$\Pr[\text{break}_6] \leq n_K \cdot \Pr[\text{break}_7]$$

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 then 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 the $\mathcal{A}$ from winning, since neither sets *win* to 1. We therefore find:

$$\Pr[\text{break}_7] \leq \mathsf{Adv}^{\mathsf{auth}}_{AEAD,\mathcal{A}'}(1^\lambda)$$

Summarizing these losses we find the adversarial advantage stated in Theorems 2-5. □

Theorem 6. Π *achieves confidentiality in stage #I if either of the responders and the static key of the initiator is uncorrupted and the responder is an honest party with:* $\mathsf{Adv}^{fACCE.conf_{\mathcal{I}}}_{\Pi,\mathcal{A}}(1^\lambda) \leq \frac{4(n_P \cdot n_S)^2}{2^\lambda} + \mathsf{Adv}^{\mathsf{CollRes}}_{\mathsf{H},\mathcal{A}'}(1^\lambda) + n_P^2 \cdot n_S^2 \cdot (\mathsf{Adv}^{SEEC}_{\Sigma,\mathcal{A}'}(1^\lambda) + \mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{IKEM,\mathcal{A}',n_S}(1^\lambda) + \mathsf{Adv}^{\mathsf{PRHO}}_{H,\mathcal{A}'}(1^\lambda) + (n_S - 1) \cdot \mathsf{Adv}^{\mathsf{auth}}_{AEAD,\mathcal{A}'}(1^\lambda) + n_K \cdot \mathsf{Adv}^{\mathsf{IND\$\text{-}CPA}}_{AEAD,\mathcal{A}'}(1^\lambda))$

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 7. Π′ *achieves confidentiality in stage #I if the responders ephemeral key and the static key of the initiator are uncorrupted and the responder is an honest party with:* $\mathsf{Adv}^{fACCE.conf_{\mathcal{I}}}_{\Pi,\mathcal{A}}(1^\lambda) \leq \frac{4(n_P \cdot n_S)^2}{2^\lambda} + \mathsf{Adv}^{\mathsf{CollRes}}_{\mathsf{H},\mathcal{A}'}(1^\lambda) + n_P^2 \cdot n_S^2 \cdot (\mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{IKEM,\mathcal{A}',n_S}(1^\lambda) + \mathsf{Adv}^{\mathsf{PRHO}}_{H,\mathcal{A}'}(1^\lambda) + (n_S - 1) \cdot \mathsf{Adv}^{\mathsf{auth}}_{AEAD,\mathcal{A}'}(1^\lambda) + n_K \cdot \mathsf{Adv}^{\mathsf{IND\$\text{-}CPA}}_{AEAD,\mathcal{A}'}(1^\lambda))$

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 8. Π *achieves confidentiality in stage #R if either of the initiators keys and the static key of the responder is uncorrupted and the initiator is an honest party with:* $\mathsf{Adv}^{fACCE.conf_{\mathcal{R}}}_{\Pi,\mathcal{A}}(1^\lambda) \leq \frac{4(n_P \cdot n_S)^2}{2^\lambda} + \mathsf{Adv}^{\mathsf{CollRes}}_{\mathsf{H},\mathcal{A}'}(1^\lambda) + n_P^2 \cdot n_S^2 \cdot (\mathsf{Adv}^{SEEC}_{\Sigma,\mathcal{A}'}(1^\lambda) + \mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{RKEM,\mathcal{A}',n_S}(1^\lambda) + \mathsf{Adv}^{\mathsf{PRHO}}_{H,\mathcal{A}'}(1^\lambda) + (n_S - 1) \cdot \mathsf{Adv}^{\mathsf{auth}}_{AEAD,\mathcal{A}'}(1^\lambda) + n_K \cdot \mathsf{Adv}^{\mathsf{IND\$\text{-}CPA}}_{AEAD,\mathcal{A}'}(1^\lambda))$

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 9. Π′ *achieves confidentiality in stage #R if the initiators ephemeral key and the static key of the responder are uncorrupted and the initiator is an honest party with:* $\mathsf{Adv}^{fACCE.conf_{\mathcal{R}}}_{\Pi,\mathcal{A}}(1^\lambda) \leq \frac{4(n_P \cdot n_S)^2}{2^\lambda} + \mathsf{Adv}^{\mathsf{CollRes}}_{\mathsf{H},\mathcal{A}'}(1^\lambda) + n_P^2 \cdot n_S^2 \cdot (\mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{RKEM,\mathcal{A}',n_S}(1^\lambda) + \frac{4(n_P \cdot n_S)^2}{2^\lambda} + \mathsf{Adv}^{\mathsf{PRHO}}_{H,\mathcal{A}'}(1^\lambda) + (n_S - 1) \cdot \mathsf{Adv}^{\mathsf{auth}}_{AEAD,\mathcal{A}'}(1^\lambda) + n_K \cdot \mathsf{Adv}^{\mathsf{IND\$\text{-}CPA}}_{AEAD,\mathcal{A}'}(1^\lambda))$

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.

Proof. We use game hoping 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 6 and 7, $\mathcal{B}$ would be the Responder and $\mathcal{C}$ would be the Initiator, and, for Theorem 8 and 9, 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 IND-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:

$$\Pr[\text{break}_4] \leq \Pr[\text{break}_5] + \mathsf{Adv}^{\text{IND-CCA}}_{XKEM,\mathcal{A}',n_S}(1^\lambda)$$

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 $\mathsf{Rand}(i_{\mathsf{HO}})$ 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:

$$\Pr[\text{break}_{7.0}] \leq (n_S - 1) \cdot \Pr[\text{break}_{7.1}]$$

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:

$$\Pr[\text{break}_{7.1}] \leq \Pr[\text{break}_{7.2}] + \mathsf{Adv}^{\text{auth}}_{AEAD,\mathcal{A}'}(1^\lambda)$$

Combining the sub-games we get:

$$\Pr[\text{break}_{7.0}] \leq \Pr[\text{break}_{7.2}] + (n_S - 1) \cdot \mathsf{Adv}^{\text{auth}}_{AEAD,\mathcal{A}'}(1^\lambda)$$

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:

$$\Pr[\text{break}_8] = \Pr[\text{break}_9] = \mathsf{Adv}^{\mathsf{IND\$\text{-}CPA}}_{AEAD,\mathcal{A}'}(1^\lambda)$$

Summarizing these losses we find the adversarial advantage. □

Theorem 10. 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:*

$\mathsf{Adv}^{fACCE.conf_E}_{\Pi,\mathcal{A}}(1^\lambda) \leq \frac{4(n_P \cdot n_S)^2}{2^\lambda} + \mathsf{Adv}^{\mathsf{CollRes}}_{\mathsf{H},\mathcal{A}'}(1^\lambda) + n_P^2 \cdot n_S^2 \cdot (2 \cdot \mathsf{Adv}^{SEEC}_{\Sigma,\mathcal{A}'}(1^\lambda) + \mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{EKEM,\mathcal{A}',1}(1^\lambda) + \mathsf{Adv}^{\mathsf{PRHO}}_{H,\mathcal{A}'}(1^\lambda) + n_K \cdot \mathsf{Adv}^{\mathsf{IND\$\text{-}CPA}}_{AEAD,\mathcal{A}'}(1^\lambda))$

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 11. Π′ *achieves confidentiality in stage #E if neither the initiators nor the responders ephemeral keys are uncorrupted and both are honest partners with:*

$\mathsf{Adv}^{fACCE.conf_E}_{\Pi,\mathcal{A}}(1^\lambda) \leq \frac{4(n_P \cdot n_S)^2}{2^\lambda} + \mathsf{Adv}^{\mathsf{CollRes}}_{\mathsf{H},\mathcal{A}'}(1^\lambda) + n_P^2 \cdot n_S^2 \cdot (\mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{EKEM,\mathcal{A}',1}(1^\lambda) + \mathsf{Adv}^{\mathsf{PRHO}}_{H,\mathcal{A}'}(1^\lambda) + n_K \cdot \mathsf{Adv}^{\mathsf{IND\$\text{-}CPA}}_{AEAD,\mathcal{A}'}(1^\lambda))$

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.

Proof. We use game hoping 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 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 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 trivialy 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 trivialy 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-1CCA-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 $\mathcal{I}$ 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:

$$\Pr[\text{break}_4] \leq \Pr[\text{break}_5] + \mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{EKEM,\mathcal{A}',1}(1^\lambda)$$

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 input/finalize functions of H she will instead invoke the In/Fin oracle, except: Whenever $\mathcal{C}$ would normally use the input/finalize functions of H with the previously replaced value k_E, she will instead invoke 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 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 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:

$$\Pr[\text{break}_7] = \Pr[\text{break}_8] = \mathsf{Adv}^{\mathsf{IND\$\text{-}CPA}}_{AEAD,\mathcal{A}'}(1^\lambda)$$

Summarizing these losses we find the adversarial advantage stated in Theorems 10 and 11. □

B. Standard Definitions

B.1. AEAD

We essentially use the same definition for AEAD that was used by the fACCE-paper [DRS20] which covers both unforgability and indistinguishability in one notion, except that we fix two minor aspects in the definition: First we ban nonce-reuse and require that $|m_0| = |m_1|$. Secondly we swap the cases of $b = 0$ and $b = 1$ to be more consistent with our other definitions (which use 0 for the "real" and 1 for the "idealized" case). In addition to that we remove an unnecessary branch from the Dec-oracle to simplify the experiment (this is a purely cosmetic change).

Definition 4 (AEAD)**.** An AEAD-scheme is a tuple of three algorithms:

- Gen is a probabilistic algorithm that takes a security-parameter 1^λ and returns a key k from a keyspace $\mathcal{K}$.
- Enc takes a key k, a nonce ň 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, a nonce ň, a ciphertext c and associated data ad and returns a message m.

Definition 5 (AEAD-completeness)**.** An AEAD-scheme *AEAD* is perfectly complete if:

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

Definition 6 (AEAD-confidentiality)**.** An AEAD-scheme *AEAD* is IND$-CPA-secure if:

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

where $\mathcal{A}$ is not allowed to query the oracle with the same nonce twice and $ is an oracle that returns randomness on all queries.

Definition 7 (AEAD-authenticity)**.** An AEAD-scheme *AEAD* offers authenticity if:

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

Where $\mathsf{Exp}^{auth}_{AEAD,\mathcal{A}}$ is defined as in Experiment 2.

Experiment 2: $\mathsf{Exp}^{auth}_{AEAD,\mathcal{A}}$, the security experiment for an AEAD-scheme *AEAD*.

$k := \mathsf{Gen}(1^\lambda)$
$N := \emptyset$
$C := \emptyset$
Oracle $\mathsf{Enc}(n, ad, m)$**:**
  $\mathsf{abort\_if}\,(n \in N)$
  $c := \mathsf{Enc}(k, n, ad, m_0)$
  $N \cup = \{n\}$
  $C \cup = \{(n, c, ad)\}$
  return c
$n, c, ad := \mathcal{A}^{\mathsf{Enc}}(1^\lambda)$
$\mathsf{abort\_if}\,((n, c, ad) \in C)$
return $\mathsf{Dec}(k, n, c, ad) \neq \perp$

B.2. PRF

A Pseudo-Random Function (PRF) is a function that takes two arguments k and m and returns a pseudo-random string *rand*.

Definition 8 (PRF)**.** We say that a function H is a secure Pseudo-Random Function (PRF) if and only if:

$$\forall \mathcal{A} \in \mathrm{QPT}, \lambda \in \mathbb{N}:$$
$$\left|\Pr\left[\mathsf{Exp}^{\mathsf{PRF}}_{\mathsf{H},\mathcal{A},0}(1^\lambda) = 1\right] - \Pr\left[\mathsf{Exp}^{\mathsf{PRF}}_{\mathsf{H},\mathcal{A},1}(1^\lambda) = 1\right]\right|$$
$$=: \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{H},\mathcal{A}}(1^\lambda) \leq \mathrm{negl}(\lambda)$$

Where $\mathsf{Exp}^{\mathsf{PRF}}_{\mathsf{H},\mathcal{A}}$ is defined as in Experiment 3.

Experiment 3: $\mathsf{Exp}^{\mathsf{PRF}}_{\mathsf{H},\mathcal{A},b}$, the security experiment for a PRF H.

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

Definition 9 (PRF-SWAP)**.** We say that a function H is a secure swapped Pseudo-Random Function (PRF-SWAP) if and only if it is a secure PRF if its arguments are swapped.

Definition 10 (dual-PRF)**.** We say that a function H is a secure dual Pseudo-Random Function (dual-PRF) if and only if it is both a secure PRF and a secure PRF-SWAP.

B.3. KEMs

Definition 11 (KEM)**.** A Key -Encapsulation Mechanism (KEM) is a tuple of three algorithms:

- Gen is a probabilistic algorithm that takes a security-parameter 1^λ and returns a keypair $(pk, sk) \in \mathcal{PK} \times \mathcal{SK}$.
- Enc is a probabilistic algorithm that takes a public key $pk \in \mathcal{PK}$ and returns a ciphertext c from a ciphertext-space C and a shared secret ss from a secret-space $\mathcal{SS}$.
- Dec takes a secret key $sk \in \mathcal{SK}$ and a ciphertext $c \in \mathcal{C}$ and returns a shared secret $ss \in \mathcal{SS}$.

Definition 12 (KEM-Completeness)**.** We say that a KEM *KEM* is perfectly complete if:

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

Definition 13 (IND-CCA)**.** We say that a KEM *KEM* offers INDistinguishability under Chosen Ciphertext Attacks (IND-CCA) if and only if:

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

Where $\mathsf{Exp}^{\mathsf{IND\text{-}CCA}}_{KEM,\mathcal{A},q}$ is defined as in Experiment 4.

Intuitively q is the maximum number of oracle-queries that the adversary may perform.

Experiment 4: $\mathsf{Exp}^{\mathsf{IND\text{-}CCA}}_{KEM,\mathcal{A},q}$, the security experiment for an IND-CCA-KEM *KEM*.

$pk, sk := KEM.\mathsf{gen}(1^\lambda)$
$c^*, k_0 := KEM.\mathsf{enc}(pk)$
$queries := 0$
$k_1 \xleftarrow{\$} \{0,1\}^\lambda$
$b \xleftarrow{\$} \{0,1\}$
Oracle $\mathsf{Dec}(c)$**:**
  $queries + = 1$
  $\mathsf{abort\_if}\,(queries > q \vee c = c^*)$
  return $KEM.\mathsf{dec}(sk, c)$
$b' := \mathcal{A}^{\mathsf{Dec}}(c^*, k_b)$
return $b = b'$

C. SEEC

Definition 14 (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 GenRand that only pass sk as argument shall be considered a shorthand for passing a independently sampled random r.

Definition 15 (SEEC-security)**.** A SEEC-scheme Σ is secure if and only if:

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

Where $\mathsf{Exp}^{SEEC}_{\Sigma,\mathcal{A}}$ is defined as in Experiment 5.

Experiment 5: $\mathsf{Exp}^{SEEC}_{\Sigma,\mathcal{A}}$, the security experiment for SEEC for a scheme Σ.

$sk := \mathsf{GenKey}(1^\lambda)$
$b \xleftarrow{\$} \{0,1\}$
$randomnesses := []$
$r\_revealed := \emptyset$
$keys\_revealed := 0$
$challenges := \emptyset$
$keys := [sk]$
$i := 0$
Oracle GenRand':
  $r \xleftarrow{\$} \{0,1\}^\lambda$
  $randomnesses[i] := r$
  $i := i + 1$
  $ret, sk := \mathsf{GenRand}(sk, r)$
  $keys[i] := sk$ **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:
  abort_if $(challenge\_index \neq \perp)$
  $r \xleftarrow{\$} \{0,1\}^\lambda$
  $R_0, sk := \mathsf{GenRand}(sk, r)$
  $R_1 \xleftarrow{\$} \{0,1\}^\lambda$
  $randomnesses[i] := r$
  $challenges \cup = \{i\}$
  $i := i + 1$
  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 $b = b'$

On a high level this definition has similarities to that of a dual-PRF (see Appendix B.2), as it requires the adversary to distinguish the output of a function that receives one random and one known output 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 corrupted; since that may sometimes be considered acceptable, we provide a full analysis of both cases.

We provide PRP-SEEC in the supplementary material (Appendix E) 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).

D. 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 3.

D.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 16 (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:

- $\mathsf{KGen} \to_\$ (sk, pk)$ 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 $sk \in \mathcal{LSK} \cup \{\perp\}, ppk \in \mathcal{LPK} \cup \{\perp\}, ad \in \{0,1\}^*, st \in \mathcal{ST}$.
- $\mathsf{Send}(sk, st, m) \rightarrow_{\$} (st', c)$ continues the protocol execution in a session and takes message m to output new state st', and ciphertext c, 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 $(epk, esk) \in \mathcal{EPK} \times \mathcal{ESK}$.
- $\mathsf{Recv}(sk, st, c) \rightarrow_{\$} (st', m)$ 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.

As described in Section 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 17 (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 $(\perp, \perp)$ 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 $m^l_* = m^l_\circ$.

D.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 encapsulate 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^s_i. 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[3]. After every invocation of Send or Recv of a session π^s_i, the respective ciphertext is appended to $\pi^s_i.T_s$ or $\pi^s_i.T_r$ respectively.
- $st \in \mathcal{ST}$: All protocol-specific local variables.
- $rand \in \{0,1\}^*$: Any random coins used by π^s_i's protocol execution.
- $\pi^s_i.rr \in \{0,1\}$: A flag indicating if $\mathcal{A}$ has leaked the ephemeral randomness used during the session execution.
- $(b_1, b_2, b_3, ..., b_{n_T})$: A vector of challenge bits the adversary is to guess (one bit for each stage).
- $fr_1, fr_2, \ldots, fr_{n_T} \in \{0,1\}$: 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^s_i.

At the beginning of the game, for all sessions π^s_i the following initial values are set: $\pi^s_i.T_s$, $\pi^s_i.T_r$, $\leftarrow \epsilon$, and $\pi^s_i.\mathsf{FT} \leftarrow \mathsf{ST}$, and $\pi^s_i.fr_{\varsigma^*} \leftarrow 1$ for all $\varsigma^* \in [0, \ldots, n_T]$, and $\pi^s_i.rand \leftarrow_{\$} \{0,1\}^*$, $\pi^s_i.b_{\varsigma^*} \leftarrow_{\$} \{0,1\}$ for all $\varsigma^* \in \mathbb{N}$ 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.

[3] Note that in what follows, we refer to these messages generically as ``ciphertexts''.

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 18 (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 yet, the initiator may have multiple honest partners (if the resulting ciphertexts are forwarded to multiple sessions). Due to the last requirement in Definition 18, 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 18) 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 19 (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 *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*.

D.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 $(\vec{s}, \mathtt{cf}, \mathtt{au^i}, \mathtt{au^r})$ 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 $\mathtt{cf}$, $\mathtt{au^i}$, and $\mathtt{au^r}$ are stage counters. The intuition here is that each row describes what stages confidentiality ($\mathtt{cf}$), authentication of initiators ($\mathtt{au^i}$) and authentication of responders ($\mathtt{au^r}$) 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, $\mathtt{au}^{\mathtt{i}} = \infty$).

D.4. Adversarial Model

In order to model active attacks in our environment, the security experiment provides the $\mathsf{OInit}, \mathsf{OSend}, \mathsf{ORecv}$ 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 $j \in \{0, \ldots |\mathsf{FT}|\}$ [4] , 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 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 (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 $st' \neq \perp$), 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

[4] 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 $j \in \{0, \ldots |\mathsf{FT}|$.''

$\pi_i^s.\mathsf{FT}$ 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 $\pi_i^s.\mathsf{FT} \leftarrow \pi_i^s.\mathsf{FT} \ \{(\vec{s}_j, \mathtt{cf}_j, \mathtt{au}_j^{\mathtt{i}}, \mathtt{au}_j^{\mathtt{r}})\}$. 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[5]. 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 $c \in Rpl$, $\pi_j^t.fr_\varsigma \leftarrow 0$ for all ς.

D.5. Security Definition

Definition 20 (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 $\mathsf{Exp}^{\mathrm{fACCE,ST}}_{n_P,n_S,\mathcal{A}}(\lambda) = 1$. We define the advantage of an adversary $\mathcal{A}$ breaking a flexible ACCE protocol fACCE as $\mathsf{Adv}^{\mathrm{fACCE,ST}}_{\mathcal{A}} = \Pr[\mathsf{Exp}^{\mathrm{fACCE,ST}}_{n_P,n_S,\mathcal{A}}(\lambda) = 1]$.

Intuitively, a flexible ACCE protocol fACCE is secure if it is correct and $\mathsf{Adv}^{\mathrm{fACCE,ST}}_{\mathcal{A}}$ 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 $\mathsf{Adv}^{\mathrm{fACCE,ST}}_{\mathcal{Q}}$ is negligible for all quantum algorithms $\mathcal{Q}$ running in polynomial-time.

[5] 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.

E. PRP-SEEC

To provide a simple and efficient instantiation for SEEC that is both practical and demonstrates the feasability of our security-notion for SEEC, we introduce PRP-SEEC, defined in Algorithm 3. It can be summarized as combining a random static key with the ephemeral entropy via a pseudo random permutation PRP (as defined in Appendix E.1), 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 uncorrupted ephemeral randomness and may be able to use existing hardware-acceleration for specific PRP-schemes, such as AES.

Algorithm 3: PRP-SEEC

```
Function GenKey (1^λ):
    return PRP.gen ()
Function GenRand (sk, r):
    return PRP.enc(sk, r), sk
```

Theorem 12. *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 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 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:

$$\Pr[\mathrm{break}_0] \leq \Pr[\mathrm{break}_1] + \frac{n^2}{2^\lambda}$$

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

$\mathsf{Exp}^{\mathrm{fACCE},\mathsf{ST}}_{n_P,n_S,\mathcal{A}}(\lambda)$:

win $\leftarrow 0$, $Rpl \leftarrow \emptyset$
for $i = 1$ **to** n_P **do**
  $ct_i \leftarrow 1$
  $corr_i \leftarrow 0$
  $(pk_i, sk_i) \leftarrow_\$ \mathsf{KGen}(\lambda)$
end for
$(i, s, \varsigma, b') \leftarrow_\$ \mathcal{A}^{\mathsf{OInit},\cdots}(pk_1, \ldots, pk_{n_P})$
if win $= 1$ **then**
  return 1
end if
if $\pi^s_i.fr_\varsigma = 0$ **then**
  return $b^* \leftarrow_\$ \{0,1\}$
else
  return $(b' = \pi^s_i.b_\varsigma)$
end if

$\mathsf{ORecv}(i, s, c) \rightarrow (\varsigma)$:

$(st', m) \leftarrow_\$ \mathrm{fACCE}.\mathsf{Recv}(sk_i, \pi^s_i.st, c)$
$\pi^s_i.st \leftarrow st'$
if $m \neq \perp$ **then**
  $\pi^s_i.T_r \leftarrow \pi^s_i.T_r || c$
end if
if $(\exists (j,t) : \mathsf{Part}(\pi^s_i, \pi^t_j) = 1)$ **then**
  if $(\pi^t_j.rr = 1) \wedge (\mathsf{comb} = 0)$ **then**
    for $u = 1$ **to** $|\pi^s_i.\mathsf{FT}|$ **do**
      $(\vec{s}_u, \mathtt{cf}_u, \mathtt{au}^{\mathtt{i}}_u, \mathtt{au}^{\mathtt{r}}_u) \leftarrow \pi^s_i.\mathsf{FT}[u]$
      if $(esk_j \in \vec{s}_u)$ **then**
        $\pi^s_i.\mathsf{FT}[u] \leftarrow \emptyset$
      end if
    end for
  end if
  if $(\pi^t_j.rr = 1) \wedge (corr_j) \wedge (\mathsf{comb} = 1)$ **then**
    for $u = 1$ **to** $|\pi^s_i.\mathsf{FT}|$ **do**
      $(\vec{s}_u, \mathtt{cf}_u, \mathtt{au}^{\mathtt{i}}_u, \mathtt{au}^{\mathtt{r}}_u) \leftarrow \pi^s_i.\mathsf{FT}[u]$
      if $(esk_j \in \vec{s}_u)$ **then**
        $\pi^s_i.\mathsf{FT}[u] \leftarrow \emptyset$
      end if
    end for
  end if
end if
for $\varsigma = 1$ **to** n_T **do**
  if $(\nexists\ \mathtt{cf}_u \in \pi^s_i.\mathsf{FT} : \varsigma \geq \mathtt{cf}_u)$ **then**
    $\pi^s_i.fr_\varsigma \leftarrow 0.$
  end if
end for
if $(\nexists (j,t) : \pi^t_j.T_s \supseteq \pi^s_i.T_r)$ **then**
  if $(\nexists \mathtt{au}_u^{\pi^s_i.\hat{\rho}} \in \pi^s_i.\mathsf{FT} : \mathtt{au}_u^{\pi^s_i.\hat{\rho}} \leq \pi^s_i.\varsigma) \wedge (c \notin Rpl)$ **then**
    win $\leftarrow 1$
  end if
end if
$\pi^s_i.\varsigma{+}{+}$
return $\pi^s_i.\varsigma$

$\mathsf{OInit}(i, pk_j, \rho, ad) \rightarrow s$:

if $(pk_j \notin \{pk_1, \ldots pk_{n_P}\})$ **then**
  return $\perp$
end if
$s \leftarrow ct_i$, $ct_i{+}{+}$
$\pi^s_i.\rho \leftarrow \rho$, $\pi^s_i.T_s, \pi^s_i.T_r \leftarrow \epsilon$
$\pi^s_i.pid \leftarrow j$, $\pi^s_i.ad \leftarrow ad$
for $\varsigma = 1$ **to** n_T **do**
  $\pi^s_i.fr_\varsigma \leftarrow 1$
  $\pi^s_i.rand \leftarrow_\$ \{0,1\}^*$
  $\pi^s_i.rr \leftarrow 0$
  $\pi^s_i.b_\varsigma \leftarrow_\$ \{0,1\}$
end for
$\pi^s_i.st \leftarrow_\$ \mathrm{fACCE}.\mathsf{Init}(sk_i, pk_j, \rho, ad)$
$\pi^s_i.\mathsf{FT} \leftarrow \mathsf{ST}$
if $corr_{\pi^s_i.pid} = 1$ **then**
  for $u = 1$ **to** $|\pi^s_i.\mathsf{FT}|$ **do**
    $(\vec{s}_u, \mathtt{cf}_u, \mathtt{au}^{\mathtt{i}}_u, \mathtt{au}^{\mathtt{r}}_u) \leftarrow \pi^s_i.\mathsf{FT}[u]$
    if $(sk_j \in \vec{s}_u)$ **then**
      $\pi^s_i.\mathsf{FT}[u] \leftarrow \emptyset$
    end if
  end for
end if
for $\varsigma = 1$ **to** n_T **do**
  if $(\nexists\ \mathtt{cf}_u \in \pi^s_i.\mathsf{FT} : \varsigma \geq \mathtt{cf}_u)$ **then**
    $\pi^t_j.fr_\varsigma \leftarrow 0.$
  end if
end for
return s

$\mathsf{OSend}(i, s, m_0, m_1) \rightarrow (c, \varsigma)$:

if $|m_0| \neq |m_1|$ **then**
  return $\perp$
end if
$(st', c) \leftarrow_\$ \mathrm{fACCE}.\mathsf{Send}(sk_i, \pi^s_i.st, m_b)$
$\pi^s_i.st \leftarrow st'$
if $(c \neq \perp)$ **then**
  if $(\pi^s_i.T_s = \epsilon) \wedge (\pi^s_i.\rho = \mathtt{i})$ **then**
    $Rpl \leftarrow Rpl \cup \{c\}$
  end if
end if
$\pi^s_i.T_s \leftarrow \pi^s_i.T_s || c$
return $c, \pi^s_i.\varsigma$

Figure 3: fACCE experiment for adversary $\mathcal{A}$. Note that for readability, when context is clear we use b as shorthand for $\pi^s_i.b_{\pi^s_i.\varsigma}$. Part and 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.

$\mathsf{OCorrupt}(i) \to (sk_i)$:

```
corr_i ← 1
for s = 1 to n_S do
    for u = 1 to |π_i^s.FT| do
        (s⃗_u, cf_u, au_j^i, au_j^r) ← π_i^s.FT[u]
        if (sk_i ∈ s⃗_u) then
            π_i^s.FT[u] ← ∅
        end if
    end for
end for
for j = 1 to n_P do
    for t = 1 to n_S do
        if (π_j^t.pid = i) then
            for u = 1 to |π_j^t.FT| do
                (s⃗_u, cf_u, au_j^i, au_j^r) ← π_j^t.FT[u]
                if (sk_i ∈ s⃗_u) then
                    π_i^s.FT[u] ← ∅
                end if
            end for
        end if
        for ς = 1 to n_T do
            if (∄ cf_u ∈ π_j^t.FT : ς ≥ cf_u) then
                π_j^t.fr_ς ← 0.
            end if
        end for
    end for
end for
for ς = 1 to n_T do
    if (∄ cf_u ∈ π_i^s.FT : ς ≥ cf_u) then
        π_i^s.fr_ς ← 0.
    end if
end for
if (π_i^s.rr = 1) ∧ (comb = 1) then
    for u = 1 to |π_i^s.FT| do
        (s⃗_u, cf_u, au_u^i, au_u^r) ← π_i^s.FT[u]
        if (esk_j ∈ s⃗_u) then
            π_i^s.FT[u] ← ∅
        end if
    end for
end if
return sk_i
```

$\mathsf{OReveal}(i, s) \to \pi_i^s.st$:

```
for u = 1 to |π_i^s.FT| do
    (s⃗_u, cf_u, au_j^i, au_j^r) ← π_i^s.FT[u]
    if (π_i^s.st ∈ s⃗_u) then
        π_i^s.FT[u] ← ∅
    end if
end for
for j = 1 to n_P do
    for t = 1 to n_S do
        if (PrevPart(π_i^s, π_j^t)) then
            for u = 1 to |π_j^t.FT| do
                (s⃗_u, cf_u, au_j^i, au_j^r) ← π_j^t.FT[u]
                if (π_i^s.st ∈ s⃗_u) then
                    π_i^s.FT[u] ← ∅
                end if
            end for
        end if
        for ς = 1 to n_T do
            if (∄ cf_u ∈ π_j^t.FT : ς ≥ cf_u) then
                π_j^t.fr_ς ← 0.
            end if
        end for
    end for
end for
return π_i^s.st
```

$\mathsf{ORevealRandomness}(i, s, \varsigma) \to \pi_i^s.rand$:

```
π_i^s.rr ← 1
if (comb = 0) then
    for u = 1 to |π_i^s.FT| do
        (s⃗_u, cf_u, au_u^i, au_u^r) ← π_i^s.FT[u]
        if (esk_j ∈ s⃗_u) then
            π_i^s.FT[u] ← ∅
        end if
    end for
end if
if (corr_i) ∧ (comb = 1) then
    for u = 1 to |π_i^s.FT| do
        (s⃗_u, cf_u, au_u^i, au_u^r) ← π_i^s.FT[u]
        if (esk_j ∈ s⃗_u) then
            π_i^s.FT[u] ← ∅
        end if
    end for
end if
for j = 1 to n_P do
    for t = 1 to n_S do
        if (PrevPart(π_i^s, π_j^t)) then
            for u = 1 to |π_j^t.FT| do
                (s⃗_u, cf_u, au_j^i, au_j^r) ← π_j^t.FT[u]
                if (esk_i ∈ s⃗_u) ∧ (comb = 0) then
                    π_i^s.FT[u] ← ∅
                end if
                if (corr_i) ∧ (comb) ∧ (esk_i ∈ s⃗_u) then
                    π_j^t.FT[u] ← ∅
                end if
            end for
        end if
        for ς = 1 to n_T do
            if (∄ cf_u ∈ π_j^t.FT : ς ≥ cf_u) then
                π_j^t.fr_ς ← 0.
            end if
        end for
    end for
end for
for ς = 1 to n_T do
    if (∄ cf_u ∈ π_j^t.FT : ς ≥ cf_u) then
        π_j^t.fr_ς ← 0.
    end if
end for
return π_i^s.rand
```

Figure 4: Final queries for the fACCE experiment for adversary $\mathcal{A}$.

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. □

E.1. PRPs

Definition 21 (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 22 (PRP-Completeness)**.** A PRP is perfectly complete if:

$$\forall \lambda \in \mathbb{N}, m \in \{0,1\}^n : \\ \Pr[\mathsf{Dec}(k, \mathsf{Enc}(k,m)) = m \,|\, k := \mathsf{Gen}(1^\lambda)] = 1$$

Definition 23 (IND-CPA)**.** We say that a PRP *PRP* offers INDistinguishability under Chosen Plaintext Attacks (IND-CPA) if and only if:

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

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

F. Security of classical Noise

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 peers 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 hat at least one uncorrupted secret. Because of this we had to perform some interpretation of those results that resulted in Table 4.

G. Detailed Patterns

As outlined in Section 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 4) 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 longterm static key for the parties 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 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

Table 4: 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.

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	∞	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	∞

notation to indicate actions that have to happen out of band and are assumed to successful, by the time the parties start interacting with each other.

Algorithm 4: Key-Generation

```
Function KGen():
    seec_sk ← SEEC.gen_key()
    pk, sk ← XKEM.gen()
    Publish(pk)
```

The session-initialization (Algorithm 5) 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.

Algorithm 5: initialization

```
Function Init():
    h ← H("pq**_label")
    ck ← HashObject.gen("pq**_label")
```

Calls to Send and Recv (Algorithm 6) 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 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 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 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.

Algorithm 6: Basic Send- and Receive-operations.

```
Function Send(m):
    pl ← String.new()
    buf ← String.new()
    ...
    if k_i ≠ ⊥:
        c ← AEAD.enc(k_i, pl, h, n)
        n.increment()
        buf.add(c)
        h ← H(h, c)
    else:
        buf.add(pl)
        h ← H(h, pl)
    return buf
Function Recv:
    if k_i ≠ ⊥:
        c ← buf.parse_next()
        pl ← AEAD.dec(k_0, c, h, n)
        h ← H(h, c)
        n.increment()
    else:
        pl ← buf.parse_next()
        h ← H(h, pl)
    ...
    m ← pl.parse_next()
    return m
```

Sending and receiving of public keys (Algorithm 7) 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 Algorithms 8--20. 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

Algorithm 7: Sending and Receiving public keys.

```
Function Send:
    ...
    if XKEM = EKEM:
        r ← SEEC.GenRand(seec_sk)
        pk_e, sk_e := XKEM.gen(r)
    pl.add(pk_e)
    ...
Function Recv:
    ...
    pk_e ← pl.parse_next()
    ...
```

of just one that starts with a case-distinction that selects the appropriate version. This allows us to present the 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.

H. Comparison of Noise and PQNoise

We present a detailed comparison between the classical and PQNoise patterns in Figures 5--16.

Algorithm 8: pqN

Function $\mathsf{KGen}_{\mathcal{J}}()$**:**
  $seec\_sk \leftarrow \mathsf{SEEC.gen\_key}()$
Function $\mathsf{KGen}_{\mathcal{R}}()$**:**
  $seec\_sk \leftarrow \mathsf{SEEC.gen\_key}()$
  $pk_{\mathcal{R}}, sk_{\mathcal{R}} \leftarrow \mathsf{RKem.gen}()$
  $\mathsf{Publish}(pk_{\mathcal{R}})$
Function $\mathsf{Init}_{\mathcal{J}}()$**:**
  $h \leftarrow \mathsf{H}(\text{“pqN\_label”})$
  $ck \leftarrow \mathsf{HashObject.gen}(\text{“pqN\_label”})$
Function $\mathsf{Send}_{\mathcal{J}}(m)$**:**
  $pl \leftarrow \mathsf{String.new}()$
  $buf \leftarrow \mathsf{String.new}()$
  $r \leftarrow \mathsf{SEEC.gen\_rand}(seec\_sk)$
  $ct_{\mathcal{R}}, kk_{\mathcal{R}} \leftarrow \mathsf{RKem.enc}(pk_{\mathcal{R}}, r)$
  $pl.\mathsf{add}(ct_{\mathcal{R}})$
  $h \leftarrow \mathsf{H}(h, pl)$
  $buf.\mathsf{add}(pl)$
  $pl.\mathsf{flush}()$
  $prekey_{\mathcal{J}}, prekey_{\mathcal{R}} \leftarrow \mathsf{HashObject.finalize}(ck, kk_{\mathcal{R}})$
  $k_{\mathcal{J}} \leftarrow \mathsf{AEAD.gen}(prekey_{\mathcal{J}})$
  $k_{\mathcal{R}} \leftarrow \mathsf{AEAD.gen}(prekey_{\mathcal{R}})$
  $n \leftarrow \mathsf{AEAD.Nonce.new}()$
  $pl.\mathsf{add}(m)$
  $c \leftarrow \mathsf{AEAD.enc}(k_{\mathcal{J}}, pl, h, n)$
  $n.\mathsf{increment}()$
  $buf.\mathsf{add}(c)$
  $h \leftarrow \mathsf{H}(h, c)$
  return buf
Function $\mathsf{Init}_{\mathcal{R}}()$**:**
  $h \leftarrow \mathsf{H}(\text{“pqN\_label”})$
  $ck \leftarrow \mathsf{HashObject.gen}(\text{“pqN\_label”})$
Function $\mathsf{Recv}_{\mathcal{R}}(buf)$**:**
  $pl \leftarrow buf.\mathsf{parse\_next}()$
  $h \leftarrow \mathsf{H}(h, pl)$
  $ct_{\mathcal{R}} \leftarrow pl.\mathsf{parse\_next}()$
  $kk_{\mathcal{R}} \leftarrow \mathsf{RKem.dec}(sk_{\mathcal{R}}, ct_{\mathcal{R}})$
  $prekey_{\mathcal{J}}, prekey_{\mathcal{R}} \leftarrow \mathsf{HashObject.finalize}(ck, kk_{\mathcal{R}})$
  $k_{\mathcal{J}} \leftarrow \mathsf{AEAD.gen}(prekey_{\mathcal{J}})$
  $k_{\mathcal{R}} \leftarrow \mathsf{AEAD.gen}(prekey_{\mathcal{R}})$
  $n \leftarrow \mathsf{AEAD.Nonce.new}()$
  $c \leftarrow buf.\mathsf{parse\_next}()$
  $pl \leftarrow \mathsf{AEAD.dec}(k_{\mathcal{J}}, c, h, n)$
  $h \leftarrow \mathsf{H}(h, c)$
  $n.\mathsf{increment}()$
  $m \leftarrow pl.\mathsf{parse\_next}()$
  return m

Algorithm 9: pqNN

Function $\mathsf{KGen}_{\mathcal{J}}()$**:**
  $seec\_sk \leftarrow \mathsf{SEEC.gen\_key}()$
Function $\mathsf{KGen}_{\mathcal{R}}()$**:**
  $seec\_sk \leftarrow \mathsf{SEEC.gen\_key}()$
Function $\mathsf{Init}_{\mathcal{J}}()$**:**
  $h \leftarrow \mathsf{H}(\text{“pqNN\_label”})$
  $ck \leftarrow \mathsf{HashObject.gen}(\text{“pqNN\_label”})$
Function $\mathsf{Send}_{\mathcal{J}}(m)$**:**
  $pl \leftarrow \mathsf{String.new}()$
  $buf \leftarrow \mathsf{String.new}()$
  $r \leftarrow \mathsf{SEEC.gen\_rand}(seec\_sk)$
  $pk_e, sk_e \leftarrow \mathsf{EKem.gen}(r)$
  $pl.\mathsf{add}(pk_e)$
  $pl.\mathsf{add}(m)$
  $buf.\mathsf{add}(pl)$
  $h \leftarrow \mathsf{H}(h, pl)$
  return buf
Function $\mathsf{Init}_{\mathcal{R}}()$**:**
  $h \leftarrow \mathsf{H}(\text{“pqNN\_label”})$
  $ck \leftarrow \mathsf{HashObject.gen}(\text{“pqNN\_label”})$
Function $\mathsf{Recv}_{\mathcal{R}}(buf)$**:**
  $pl \leftarrow buf.\mathsf{parse\_next}()$
  $h \leftarrow \mathsf{H}(h, pl)$
  $pk_e \leftarrow pl.\mathsf{parse\_next}()$
  $m \leftarrow pl.\mathsf{parse\_next}()$
  return m
Function $\mathsf{Send}_{\mathcal{R}}(m)$**:**
  $pl \leftarrow \mathsf{String.new}()$
  $buf \leftarrow \mathsf{String.new}()$
  $r \leftarrow \mathsf{SEEC.gen\_rand}(seec\_sk)$
  $ct_e, kk_e \leftarrow \mathsf{EKem.enc}(pk_e, r)$
  $pl.\mathsf{add}(ct_e)$
  $h \leftarrow \mathsf{H}(h, pl)$
  $buf.\mathsf{add}(pl)$
  $pl.\mathsf{flush}()$
  $prekey_{\mathcal{J}}, prekey_{\mathcal{R}} \leftarrow \mathsf{HashObject.finalize}(ck, kk_e)$
  $k_{\mathcal{J}} \leftarrow \mathsf{AEAD.gen}(prekey_{\mathcal{J}})$
  $k_{\mathcal{R}} \leftarrow \mathsf{AEAD.gen}(prekey_{\mathcal{R}})$
  $n \leftarrow \mathsf{AEAD.Nonce.new}()$
  $pl.\mathsf{add}(m)$
  $c \leftarrow \mathsf{AEAD.enc}(k_{\mathcal{R}}, pl, h, n)$
  $n.\mathsf{increment}()$
  $buf.\mathsf{add}(c)$
  $h \leftarrow \mathsf{H}(h, c)$
  return buf
Function $\mathsf{Recv}_{\mathcal{J}}(buf)$**:**
  $pl \leftarrow buf.\mathsf{parse\_next}()$
  $h \leftarrow \mathsf{H}(h, pl)$
  $ct_e \leftarrow pl.\mathsf{parse\_next}()$
  $kk_e \leftarrow \mathsf{EKem.dec}(sk_e, ct_e)$
  $prekey_{\mathcal{J}}, prekey_{\mathcal{R}} \leftarrow \mathsf{HashObject.finalize}(ck, kk_e)$
  $k_{\mathcal{J}} \leftarrow \mathsf{AEAD.gen}(prekey_{\mathcal{J}})$
  $k_{\mathcal{R}} \leftarrow \mathsf{AEAD.gen}(prekey_{\mathcal{R}})$
  $n \leftarrow \mathsf{AEAD.Nonce.new}()$
  $c \leftarrow buf.\mathsf{parse\_next}()$
  $pl \leftarrow \mathsf{AEAD.dec}(k_{\mathcal{R}}, c, h, n)$
  $h \leftarrow \mathsf{H}(h, c)$
  $n.\mathsf{increment}()$
  $m \leftarrow pl.\mathsf{parse\_next}()$
  return m

Algorithm 10: pqNK

```
1  Function KGen_J():
2      seec_sk ← SEEC.gen_key()
3  Function KGen_R():
4      seec_sk ← SEEC.gen_key()
5      pk_R, sk_R ← RKem.gen()
6      Publish(pk_R)
7  Function Init_J():
8      h ← H("pqNK_label")
9      ck ← HashObject.gen("pqNK_label")
10 Function Send_J(m):
11     pl ← String.new()
12     buf ← String.new()
13     r ← SEEC.gen_rand(seec_sk)
14     ct_R, kk_R ← RKem.enc(pk_R, r)
15     pl.add(ct_R)
16     h ← H(h, pl)
17     buf.add(pl)
18     pl.flush()
19     prekey ← ck.input(kk_R)
20     k_0 ← AEAD.gen(prekey)
21     n ← AEAD.Nonce.new()
22     r ← SEEC.gen_rand(seec_sk)
23     pk_e, sk_e ← EKem.gen(r)
24     pl.add(pk_e)
25     pl.add(m)
26     c ← AEAD.enc(k_0, pl, h, n)
27     n.increment()
28     buf.add(c)
29     h ← H(h, c)
30     return buf
31 Function Init_R():
32     h ← H("pqNK_label")
33     ck ← HashObject.gen("pqNK_label")
34 Function Recv_R(buf):
35     pl ← buf.parse_next()
36     h ← H(h, pl)
37     ct_R ← pl.parse_next()
38     kk_R ← RKem.dec(sk_R, ct_R)
39     prekey ← ck.input(kk_R)
40     k_0 ← AEAD.gen(prekey)
41     n ← AEAD.Nonce.new()
42     c ← buf.parse_next()
43     pl ← AEAD.dec(k_0, c, h, n)
44     h ← H(h, c)
45     n.increment()
46     pk_e ← pl.parse_next()
47     m ← pl.parse_next()
48     return m
49 Function Send_R(m):
50     pl ← String.new()
51     buf ← String.new()
52     r ← SEEC.gen_rand(seec_sk)
53     ct_e, kk_e ← EKem.enc(pk_e, r)
54     pl.add(ct_e)
55     h ← H(h, pl)
56     buf.add(pl)
57     pl.flush()
58     prekey_J, prekey_R ← HashObject.finalize(ck, kk_e)
59     k_J ← AEAD.gen(prekey_J)
60     k_R ← AEAD.gen(prekey_R)
61     n ← AEAD.Nonce.new()
62     pl.add(m)
63     c ← AEAD.enc(k_R, pl, h, n)
64     n.increment()
65     buf.add(c)
66     h ← H(h, c)
67     return buf
68 Function Recv_J(buf):
69     c ← buf.parse_next()
70     pl ← AEAD.dec(k_0, c, h, n)
71     h ← H(h, c)
72     n.increment()
73     ct_e ← pl.parse_next()
74     kk_e ← EKem.dec(sk_e, ct_e)
75     prekey_J, prekey_R ← HashObject.finalize(ck, kk_e)
76     k_J ← AEAD.gen(prekey_J)
77     k_R ← AEAD.gen(prekey_R)
78     n ← AEAD.Nonce.new()
79     c ← buf.parse_next()
80     pl ← AEAD.dec(k_R, c, h, n)
81     h ← H(h, c)
82     n.increment()
83     m ← pl.parse_next()
84     return m
```

Algorithm 11: pqNX

Function $\mathsf{KGen}_{\mathcal{J}}()$**:**
  $seec\_sk \leftarrow \mathsf{SEEC.gen\_key}()$

Function $\mathsf{KGen}_{\mathcal{R}}()$**:**
  $seec\_sk \leftarrow \mathsf{SEEC.gen\_key}()$
  $pk_{\mathcal{R}}, sk_{\mathcal{R}} \leftarrow \mathsf{RKem.gen}()$

Function $\mathsf{Init}_{\mathcal{J}}()$**:**
  $h \leftarrow \mathsf{H}(\text{“pqNX\_label”})$
  $ck \leftarrow \mathsf{HashObject.gen}(\text{“pqNX\_label”})$

Function $\mathsf{Send}_{\mathcal{J}}(m)$**:**
  $pl \leftarrow \mathsf{String.new}()$
  $buf \leftarrow \mathsf{String.new}()$
  $r \leftarrow \mathsf{SEEC.gen\_rand}(seec\_sk)$
  $pk_e, sk_e \leftarrow \mathsf{EKem.gen}(r)$
  $pl.\mathsf{add}(pk_e)$
  $pl.\mathsf{add}(m)$
  $buf.\mathsf{add}(pl)$
  $h \leftarrow \mathsf{H}(h, pl)$
  return buf

Function $\mathsf{Init}_{\mathcal{R}}()$**:**
  $h \leftarrow \mathsf{H}(\text{“pqNX\_label”})$
  $ck \leftarrow \mathsf{HashObject.gen}(\text{“pqNX\_label”})$

Function $\mathsf{Recv}_{\mathcal{R}}(buf)$**:**
  $pl \leftarrow buf.\mathsf{parse\_next}()$
  $h \leftarrow \mathsf{H}(h, pl)$
  $pk_e \leftarrow pl.\mathsf{parse\_next}()$
  $m \leftarrow pl.\mathsf{parse\_next}()$
  return m

Function $\mathsf{Send}_{\mathcal{R}}(m)$**:**
  $pl \leftarrow \mathsf{String.new}()$
  $buf \leftarrow \mathsf{String.new}()$
  $r \leftarrow \mathsf{SEEC.gen\_rand}(seec\_sk)$
  $ct_e, kk_e \leftarrow \mathsf{EKem.enc}(pk_e, r)$
  $pl.\mathsf{add}(ct_e)$
  $h \leftarrow \mathsf{H}(h, pl)$
  $buf.\mathsf{add}(pl)$
  $pl.\mathsf{flush}()$
  $prekey \leftarrow ck.\mathsf{input}(kk_e)$
  $k_0 \leftarrow \mathsf{AEAD.gen}(prekey)$
  $n \leftarrow \mathsf{AEAD.Nonce.new}()$
  $pl.\mathsf{add}(pk_{\mathcal{R}})$
  $pl.\mathsf{add}(m)$
  $c \leftarrow \mathsf{AEAD.enc}(k_0, pl, h, n)$
  $n.\mathsf{increment}()$
  $buf.\mathsf{add}(c)$
  $h \leftarrow \mathsf{H}(h, c)$
  return buf

Function $\mathsf{Recv}_{\mathcal{J}}(buf)$**:**
  $pl \leftarrow buf.\mathsf{parse\_next}()$
  $h \leftarrow \mathsf{H}(h, pl)$
  $ct_e \leftarrow pl.\mathsf{parse\_next}()$
  $kk_e \leftarrow \mathsf{EKem.dec}(sk_e, ct_e)$
  $prekey \leftarrow ck.\mathsf{input}(kk_e)$
  $k_0 \leftarrow \mathsf{AEAD.gen}(prekey)$
  $n \leftarrow \mathsf{AEAD.Nonce.new}()$
  $c \leftarrow buf.\mathsf{parse\_next}()$
  $pl \leftarrow \mathsf{AEAD.dec}(k_0, c, h, n)$
  $h \leftarrow \mathsf{H}(h, c)$
  $n.\mathsf{increment}()$
  $pk_{\mathcal{R}} \leftarrow pl.\mathsf{parse\_next}()$
  $m \leftarrow pl.\mathsf{parse\_next}()$
  return m

Function $\mathsf{Send}_{\mathcal{J}}(m)$**:**
  $pl \leftarrow \mathsf{String.new}()$
  $buf \leftarrow \mathsf{String.new}()$
  $r \leftarrow \mathsf{SEEC.gen\_rand}(seec\_sk)$
  $ct_{\mathcal{R}}, kk_{\mathcal{R}} \leftarrow \mathsf{RKem.enc}(pk_{\mathcal{R}}, r)$
  $pl.\mathsf{add}(ct_{\mathcal{R}})$
  $c \leftarrow \mathsf{AEAD.enc}(k_0, pl, h, n)$
  $n.\mathsf{increment}()$
  $h \leftarrow \mathsf{H}(h, c)$
  $buf.\mathsf{add}(c)$
  $pl.\mathsf{flush}()$
  $prekey_{\mathcal{J}}, prekey_{\mathcal{R}} \leftarrow \mathsf{HashObject.finalize}(ck, kk_{\mathcal{R}})$
  $k_{\mathcal{J}} \leftarrow \mathsf{AEAD.gen}(prekey_{\mathcal{J}})$
  $k_{\mathcal{R}} \leftarrow \mathsf{AEAD.gen}(prekey_{\mathcal{R}})$
  $n \leftarrow \mathsf{AEAD.Nonce.new}()$
  $pl.\mathsf{add}(m)$
  $c \leftarrow \mathsf{AEAD.enc}(k_{\mathcal{J}}, pl, h, n)$
  $n.\mathsf{increment}()$
  $buf.\mathsf{add}(c)$
  $h \leftarrow \mathsf{H}(h, c)$
  return buf

Function $\mathsf{Recv}_{\mathcal{R}}(buf)$**:**
  $c \leftarrow buf.\mathsf{parse\_next}()$
  $pl \leftarrow \mathsf{AEAD.dec}(k_0, c, h, n)$
  $h \leftarrow \mathsf{H}(h, c)$
  $n.\mathsf{increment}()$
  $ct_{\mathcal{R}} \leftarrow pl.\mathsf{parse\_next}()$
  $kk_{\mathcal{R}} \leftarrow \mathsf{RKem.dec}(sk_{\mathcal{R}}, ct_{\mathcal{R}})$
  $prekey_{\mathcal{J}}, prekey_{\mathcal{R}} \leftarrow \mathsf{HashObject.finalize}(ck, kk_{\mathcal{R}})$
  $k_{\mathcal{J}} \leftarrow \mathsf{AEAD.gen}(prekey_{\mathcal{J}})$
  $k_{\mathcal{R}} \leftarrow \mathsf{AEAD.gen}(prekey_{\mathcal{R}})$
  $n \leftarrow \mathsf{AEAD.Nonce.new}()$
  $c \leftarrow buf.\mathsf{parse\_next}()$
  $pl \leftarrow \mathsf{AEAD.dec}(k_{\mathcal{J}}, c, h, n)$
  $h \leftarrow \mathsf{H}(h, c)$
  $n.\mathsf{increment}()$
  $m \leftarrow pl.\mathsf{parse\_next}()$
  return m

Algorithm 12: pqKN

Function $\mathsf{KGen}_\mathcal{I}()$:
  $seec\_sk \leftarrow \mathsf{SEEC.gen\_key}()$
  $pk_\mathcal{I}, sk_\mathcal{I} \leftarrow \mathsf{IKem.gen}()$
  $\mathsf{Publish}(pk_\mathcal{I})$

Function $\mathsf{KGen}_\mathcal{R}()$:
  $seec\_sk \leftarrow \mathsf{SEEC.gen\_key}()$

Function $\mathsf{Init}_\mathcal{I}()$:
  $h \leftarrow \mathsf{H}(\text{“pqKN\_label”})$
  $ck \leftarrow \mathsf{HashObject.gen}(\text{“pqKN\_label”})$

Function $\mathsf{Send}_\mathcal{I}(m)$:
  $pl \leftarrow \mathsf{String.new}()$
  $buf \leftarrow \mathsf{String.new}()$
  $r \leftarrow \mathsf{SEEC.gen\_rand}(seec\_sk)$
  $pk_e, sk_e \leftarrow \mathsf{EKem.gen}(r)$
  $pl.\mathsf{add}(pk_e)$
  $pl.\mathsf{add}(m)$
  $buf.\mathsf{add}(pl)$
  $h \leftarrow \mathsf{H}(h, pl)$
  return buf

Function $\mathsf{Init}_\mathcal{R}()$:
  $h \leftarrow \mathsf{H}(\text{“pqKN\_label”})$
  $ck \leftarrow \mathsf{HashObject.gen}(\text{“pqKN\_label”})$

Function $\mathsf{Recv}_\mathcal{R}(buf)$:
  $pl \leftarrow buf.\mathsf{parse\_next}()$
  $h \leftarrow \mathsf{H}(h, pl)$
  $pk_e \leftarrow pl.\mathsf{parse\_next}()$
  $m \leftarrow pl.\mathsf{parse\_next}()$
  return m

Function $\mathsf{Send}_\mathcal{R}(m)$:
  $pl \leftarrow \mathsf{String.new}()$
  $buf \leftarrow \mathsf{String.new}()$
  $r \leftarrow \mathsf{SEEC.gen\_rand}(seec\_sk)$
  $ct_e, kk_e \leftarrow \mathsf{EKem.enc}(pk_e, r)$
  $pl.\mathsf{add}(ct_e)$
  $h \leftarrow \mathsf{H}(h, pl)$
  $buf.\mathsf{add}(pl)$
  $pl.\mathsf{flush}()$
  $prekey \leftarrow ck.\mathsf{input}(kk_e)$
  $k_0 \leftarrow \mathsf{AEAD.gen}(prekey)$
  $n \leftarrow \mathsf{AEAD.Nonce.new}()$
  $r \leftarrow \mathsf{SEEC.gen\_rand}(seec\_sk)$
  $ct_\mathcal{I}, kk_\mathcal{I} \leftarrow \mathsf{IKem.enc}(pk_\mathcal{I}, r)$
  $pl.\mathsf{add}(ct_\mathcal{I})$
  $c \leftarrow \mathsf{AEAD.enc}(k_0, pl, h, n)$
  $n.\mathsf{increment}()$
  $h \leftarrow \mathsf{H}(h, c)$
  $buf.\mathsf{add}(c)$
  $pl.\mathsf{flush}()$
  $prekey_\mathcal{I}, prekey_\mathcal{R} \leftarrow \mathsf{HashObject.finalize}(ck, kk_\mathcal{I})$
  $k_\mathcal{I} \leftarrow \mathsf{AEAD.gen}(prekey_\mathcal{I})$
  $k_\mathcal{R} \leftarrow \mathsf{AEAD.gen}(prekey_\mathcal{R})$
  $n \leftarrow \mathsf{AEAD.Nonce.new}()$
  $c \leftarrow \mathsf{AEAD.enc}(k_\mathcal{R}, pl, h, n)$
  $n.\mathsf{increment}()$
  $buf.\mathsf{add}(c)$
  $h \leftarrow \mathsf{H}(h, c)$
  return buf

Function $\mathsf{Recv}_\mathcal{I}(buf)$:
  $pl \leftarrow buf.\mathsf{parse\_next}()$
  $h \leftarrow \mathsf{H}(h, pl)$
  $ct_e \leftarrow pl.\mathsf{parse\_next}()$
  $kk_e \leftarrow \mathsf{EKem.dec}(sk_e, ct_e)$
  $prekey \leftarrow ck.\mathsf{input}(kk_e)$
  $k_0 \leftarrow \mathsf{AEAD.gen}(prekey)$
  $n \leftarrow \mathsf{AEAD.Nonce.new}()$
  $c \leftarrow buf.\mathsf{parse\_next}()$
  $pl \leftarrow \mathsf{AEAD.dec}(k_0, c, h, n)$
  $h \leftarrow \mathsf{H}(h, c)$
  $n.\mathsf{increment}()$
  $ct_\mathcal{I} \leftarrow pl.\mathsf{parse\_next}()$
  $kk_\mathcal{I} \leftarrow \mathsf{IKem.dec}(sk_\mathcal{I}, ct_\mathcal{I})$
  $prekey_\mathcal{I}, prekey_\mathcal{R} \leftarrow \mathsf{HashObject.finalize}(ck, kk_\mathcal{I})$
  $k_\mathcal{I} \leftarrow \mathsf{AEAD.gen}(prekey_\mathcal{I})$
  $k_\mathcal{R} \leftarrow \mathsf{AEAD.gen}(prekey_\mathcal{R})$
  $n \leftarrow \mathsf{AEAD.Nonce.new}()$
  $c \leftarrow buf.\mathsf{parse\_next}()$
  $pl \leftarrow \mathsf{AEAD.dec}(k_\mathcal{R}, c, h, n)$
  $h \leftarrow \mathsf{H}(h, c)$
  $n.\mathsf{increment}()$
  $m \leftarrow pl.\mathsf{parse\_next}()$
  return m

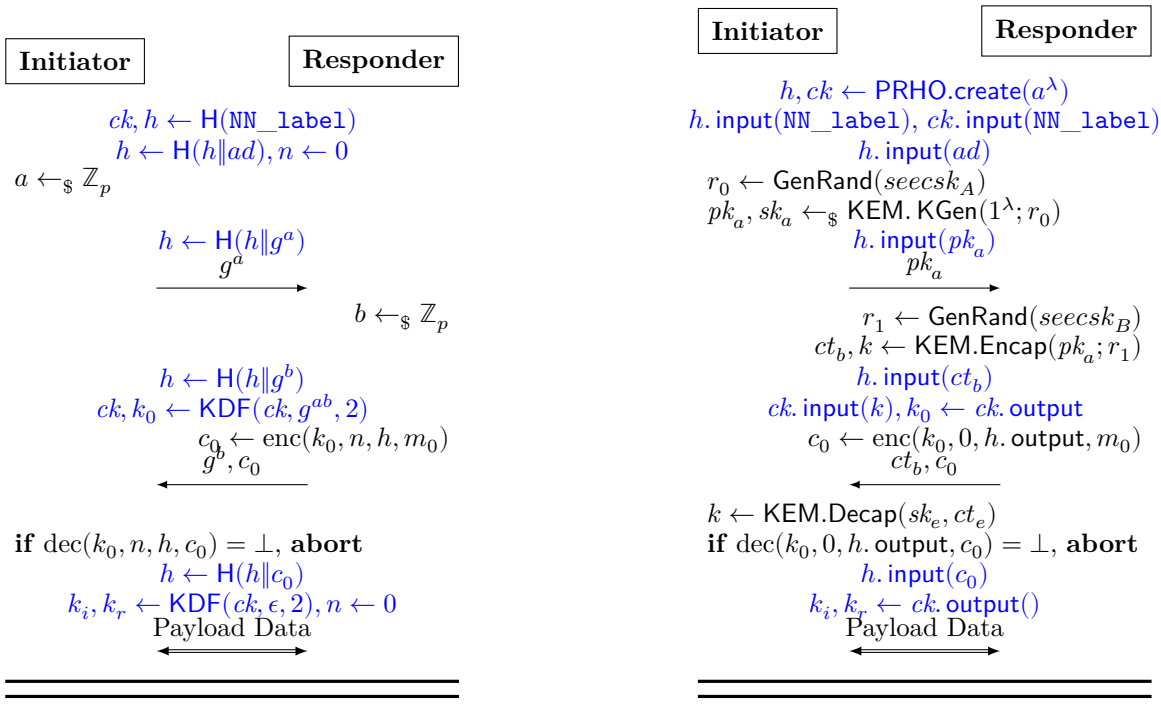

Figure 5: The NN patterns of classical Noise (left) and PQNoise (right). For reasons of space we use the following conventions here: highlighted actions are performed by both parties as soon as they receive all necessary values to perform the computations in question. If any decryption- or decapsulation-algorithm returns $\perp$, the party in question aborts the protocol.

Algorithm 13: pqKK

```
1  Function KGen_I():
2      seec_sk ← SEEC.gen_key()
3      pk_I, sk_I ← IKem.gen()
4      Publish(pk_I)
5  Function KGen_R():
6      seec_sk ← SEEC.gen_key()
7      pk_R, sk_R ← RKem.gen()
8      Publish(pk_R)
9  Function Init_I():
10     h ← H("pqKK_label")
11     ck ← HashObject.gen(
         "pqKK_label")
12 Function Send_I(m):
13     pl ← String.new()
14     buf ← String.new()
15     r ← SEEC.gen_rand(seec_sk)
16     ct_R, kk_R ← RKem.enc(pk_R, r)
17     pl.add(ct_R)
18     h ← H(h, pl)
19     buf.add(pl)
20     pl.flush()
21     prekey ← ck.input(kk_R)
22     k_0 ← AEAD.gen(prekey)
23     n ← AEAD.Nonce.new()
24     r ← SEEC.gen_rand(seec_sk)
25     pk_e, sk_e ← EKem.gen(r)
26     pl.add(pk_e)
27     pl.add(m)
28     c ← AEAD.enc(k_0, pl, h, n)
29     n.increment()
30     buf.add(c)
31     h ← H(h, c)
32     return buf
33 Function Init_R():
34     h ← H("pqKK_label")
35     ck ← HashObject.gen(
         "pqKK_label")
36 Function Recv_R(buf):
37     pl ← buf.parse_next()
38     h ← H(h, pl)
39     ct_R ← pl.parse_next()
40     kk_R ← RKem.dec(sk_R, ct_R)
41     prekey ← ck.input(kk_R)
42     k_0 ← AEAD.gen(prekey)
43     n ← AEAD.Nonce.new()
44     c ← buf.parse_next()
45     pl ← AEAD.dec(k_0, c, h, n)
46     h ← H(h, c)
47     n.increment()
48     pk_e ← pl.parse_next()
49     m ← pl.parse_next()
50     return m
51 Function Send_R(m):
52     pl ← String.new()
53     buf ← String.new()
54     r ← SEEC.gen_rand(seec_sk)
55     ct_e, kk_e ← EKem.enc(pk_e, r)
56     pl.add(ct_e)
57     h ← H(h, pl)
58     buf.add(pl)
59     pl.flush()
60     prekey ← ck.input(kk_e)
61     k_1 ← AEAD.gen(prekey)
62     n ← AEAD.Nonce.new()
63     r ← SEEC.gen_rand(seec_sk)
64     ct_I, kk_I ← IKem.enc(pk_I, r)
65     pl.add(ct_I)
66     c ← AEAD.enc(k_1, pl, h, n)
67     n.increment()
68     h ← H(h, c)
69     buf.add(c)
70     pl.flush()
71     prekey_I, prekey_R ←
         HashObject.finalize(ck, kk_I)
72     k_I ← AEAD.gen(prekey_I)
73     k_R ← AEAD.gen(prekey_R)
74     n ← AEAD.Nonce.new()
75     pl.add(m)
76     c ← AEAD.enc(k_R, pl, h, n)
77     n.increment()
78     buf.add(c)
79     h ← H(h, c)
80     return buf
81 Function Recv_I(buf):
82     c ← buf.parse_next()
83     pl ← AEAD.dec(k_0, c, h, n)
84     h ← H(h, c)
85     n.increment()
86     ct_e ← pl.parse_next()
87     kk_e ← EKem.dec(sk_e, ct_e)
88     prekey ← ck.input(kk_e)
89     k_1 ← AEAD.gen(prekey)
90     n ← AEAD.Nonce.new()
91     c ← buf.parse_next()
92     pl ← AEAD.dec(k_0, c, h, n)
93     h ← H(h, c)
94     n.increment()
95     ct_I ← pl.parse_next()
96     kk_I ← IKem.dec(sk_I, ct_I)
97     prekey_I, prekey_R ←
         HashObject.finalize(ck, kk_I)
98     k_I ← AEAD.gen(prekey_I)
99     k_R ← AEAD.gen(prekey_R)
100    n ← AEAD.Nonce.new()
101    c ← buf.parse_next()
102    pl ← AEAD.dec(k_R, c, h, n)
103    h ← H(h, c)
104    n.increment()
105    m ← pl.parse_next()
106    return m
```

Algorithm 14: pqKX

```
Function KGen_I():
    seec_sk ← SEEC.gen_key()
    pk_I, sk_I ← IKem.gen()
    Publish(pk_I)
Function KGen_R():
    seec_sk ← SEEC.gen_key()
    pk_R, sk_R ← RKem.gen()
Function Init_I():
    h ← H("pqKX_label")
    ck ← HashObject.gen("pqKX_label")
Function Send_I(m):
    pl ← String.new()
    buf ← String.new()
    r ← SEEC.gen_rand(seec_sk)
    pk_e, sk_e ← EKem.gen(r)
    pl.add(pk_e)
    pl.add(m)
    buf.add(pl)
    h ← H(h, pl)
    return buf
Function Init_R():
    h ← H("pqKX_label")
    ck ← HashObject.gen("pqKX_label")
Function Recv_R(buf):
    pl ← buf.parse_next()
    h ← H(h, pl)
    pk_e ← pl.parse_next()
    m ← pl.parse_next()
    return m
Function Send_R(m):
    pl ← String.new()
    buf ← String.new()
    r ← SEEC.gen_rand(seec_sk)
    ct_e, kk_e ← EKem.enc(pk_e, r)
    pl.add(ct_e)
    h ← H(h, pl)
    buf.add(pl)
    pl.flush()
    prekey ← ck.input(kk_e)
    k_0 ← AEAD.gen(prekey)
    n ← AEAD.Nonce.new()
    r ← SEEC.gen_rand(seec_sk)
    ct_I, kk_I ← IKem.enc(pk_I, r)
    pl.add(ct_I)
    c ← AEAD.enc(k_0, pl, h, n)
    n.increment()
    h ← H(h, c)
    buf.add(c)
    pl.flush()
    prekey ← ck.input(kk_I)
    k_1 ← AEAD.gen(prekey)
    n ← AEAD.Nonce.new()
    pl.add(pk_R)
    pl.add(m)
    c ← AEAD.enc(k_1, pl, h, n)
    n.increment()
    buf.add(c)
    h ← H(h, c)
    return buf
Function Recv_I(buf):
    pl ← buf.parse_next()
    h ← H(h, pl)
    ct_e ← pl.parse_next()
    kk_e ← EKem.dec(sk_e, ct_e)
    prekey ← ck.input(kk_e)
    k_0 ← AEAD.gen(prekey)
    n ← AEAD.Nonce.new()
    c ← buf.parse_next()
    pl ← AEAD.dec(k_0, c, h, n)
    h ← H(h, c)
    n.increment()
    ct_I ← pl.parse_next()
    kk_I ← IKem.dec(sk_I, ct_I)
    prekey ← ck.input(kk_I)
    k_1 ← AEAD.gen(prekey)
    n ← AEAD.Nonce.new()
    c ← buf.parse_next()
    pl ← AEAD.dec(k_0, c, h, n)
    h ← H(h, c)
    n.increment()
    pk_R ← pl.parse_next()
    m ← pl.parse_next()
    return m
Function Send_I(m):
    pl ← String.new()
    buf ← String.new()
    r ← SEEC.gen_rand(seec_sk)
    ct_R, kk_R ← RKem.enc(pk_R, r)
    pl.add(ct_R)
    c ← AEAD.enc(k_1, pl, h, n)
    n.increment()
    h ← H(h, c)
    buf.add(c)
    pl.flush()
    prekey_I, prekey_R ← HashObject.finalize(ck, kk_R)
    k_I ← AEAD.gen(prekey_I)
    k_R ← AEAD.gen(prekey_R)
    n ← AEAD.Nonce.new()
    pl.add(m)
    c ← AEAD.enc(k_I, pl, h, n)
    n.increment()
    buf.add(c)
    h ← H(h, c)
    return buf
Function Recv_R(buf):
    c ← buf.parse_next()
    pl ← AEAD.dec(k_1, c, h, n)
    h ← H(h, c)
    n.increment()
    ct_R ← pl.parse_next()
    kk_R ← RKem.dec(sk_R, ct_R)
    prekey_I, prekey_R ← HashObject.finalize(ck, kk_R)
    k_I ← AEAD.gen(prekey_I)
    k_R ← AEAD.gen(prekey_R)
    n ← AEAD.Nonce.new()
    c ← buf.parse_next()
    pl ← AEAD.dec(k_I, c, h, n)
    h ← H(h, c)
    n.increment()
    m ← pl.parse_next()
    return m
```

Algorithm 15: pqXN

Function $\mathsf{KGen}_{\mathcal{J}}()$**:**
  $seec\_sk \leftarrow \mathsf{SEEC.gen\_key}()$
  $pk_{\mathcal{J}}, sk_{\mathcal{J}} \leftarrow \mathsf{IKem.gen}()$

Function $\mathsf{KGen}_{\mathcal{R}}()$**:**
  $seec\_sk \leftarrow \mathsf{SEEC.gen\_key}()$

Function $\mathsf{Init}_{\mathcal{J}}()$**:**
  $h \leftarrow \mathsf{H}(\text{“pqXN\_label”})$
  $ck \leftarrow \mathsf{HashObject.gen}(\text{“pqXN\_label”})$

Function $\mathsf{Send}_{\mathcal{J}}(m)$**:**
  $pl \leftarrow \mathsf{String.new}()$
  $buf \leftarrow \mathsf{String.new}()$
  $r \leftarrow \mathsf{SEEC.gen\_rand}(seec\_sk)$
  $pk_e, sk_e \leftarrow \mathsf{EKem.gen}(r)$
  $pl.\mathsf{add}(pk_e)$
  $pl.\mathsf{add}(m)$
  $buf.\mathsf{add}(pl)$
  $h \leftarrow \mathsf{H}(h, pl)$
  return buf

Function $\mathsf{Init}_{\mathcal{R}}()$**:**
  $h \leftarrow \mathsf{H}(\text{“pqXN\_label”})$
  $ck \leftarrow \mathsf{HashObject.gen}(\text{“pqXN\_label”})$

Function $\mathsf{Recv}_{\mathcal{R}}(buf)$**:**
  $pl \leftarrow buf.\mathsf{parse\_next}()$
  $h \leftarrow \mathsf{H}(h, pl)$
  $pk_e \leftarrow pl.\mathsf{parse\_next}()$
  $m \leftarrow pl.\mathsf{parse\_next}()$
  return m

Function $\mathsf{Send}_{\mathcal{R}}(m)$**:**
  $pl \leftarrow \mathsf{String.new}()$
  $buf \leftarrow \mathsf{String.new}()$
  $r \leftarrow \mathsf{SEEC.gen\_rand}(seec\_sk)$
  $ct_e, kk_e \leftarrow \mathsf{EKem.enc}(pk_e, r)$
  $pl.\mathsf{add}(ct_e)$
  $h \leftarrow \mathsf{H}(h, pl)$
  $buf.\mathsf{add}(pl)$
  $pl.\mathsf{flush}()$
  $prekey \leftarrow ck.\mathsf{input}(kk_e)$
  $k_0 \leftarrow \mathsf{AEAD.gen}(prekey)$
  $n \leftarrow \mathsf{AEAD.Nonce.new}()$
  $pl.\mathsf{add}(m)$
  $c \leftarrow \mathsf{AEAD.enc}(k_0, pl, h, n)$
  $n.\mathsf{increment}()$
  $buf.\mathsf{add}(c)$
  $h \leftarrow \mathsf{H}(h, c)$
  return buf

Function $\mathsf{Recv}_{\mathcal{J}}(buf)$**:**
  $pl \leftarrow buf.\mathsf{parse\_next}()$
  $h \leftarrow \mathsf{H}(h, pl)$
  $ct_e \leftarrow pl.\mathsf{parse\_next}()$
  $kk_e \leftarrow \mathsf{EKem.dec}(sk_e, ct_e)$
  $prekey \leftarrow ck.\mathsf{input}(kk_e)$
  $k_0 \leftarrow \mathsf{AEAD.gen}(prekey)$
  $n \leftarrow \mathsf{AEAD.Nonce.new}()$
  $c \leftarrow buf.\mathsf{parse\_next}()$
  $pl \leftarrow \mathsf{AEAD.dec}(k_0, c, h, n)$
  $h \leftarrow \mathsf{H}(h, c)$
  $n.\mathsf{increment}()$
  $m \leftarrow pl.\mathsf{parse\_next}()$
  return m

Function $\mathsf{Send}_{\mathcal{J}}(m)$**:**
  $pl \leftarrow \mathsf{String.new}()$
  $buf \leftarrow \mathsf{String.new}()$
  $pl.\mathsf{add}(pk_{\mathcal{J}})$
  $pl.\mathsf{add}(m)$
  $c \leftarrow \mathsf{AEAD.enc}(k_0, pl, h, n)$
  $n.\mathsf{increment}()$
  $buf.\mathsf{add}(c)$
  $h \leftarrow \mathsf{H}(h, c)$
  return buf

Function $\mathsf{Recv}_{\mathcal{R}}(buf)$**:**
  $c \leftarrow buf.\mathsf{parse\_next}()$
  $pl \leftarrow \mathsf{AEAD.dec}(k_0, c, h, n)$
  $h \leftarrow \mathsf{H}(h, c)$
  $n.\mathsf{increment}()$
  $pk_{\mathcal{J}} \leftarrow pl.\mathsf{parse\_next}()$
  $m \leftarrow pl.\mathsf{parse\_next}()$
  return m

Function $\mathsf{Send}_{\mathcal{R}}(m)$**:**
  $pl \leftarrow \mathsf{String.new}()$
  $buf \leftarrow \mathsf{String.new}()$
  $r \leftarrow \mathsf{SEEC.gen\_rand}(seec\_sk)$
  $ct_{\mathcal{J}}, kk_{\mathcal{J}} \leftarrow \mathsf{IKem.enc}(pk_{\mathcal{J}}, r)$
  $pl.\mathsf{add}(ct_{\mathcal{J}})$
  $c \leftarrow \mathsf{AEAD.enc}(k_0, pl, h, n)$
  $n.\mathsf{increment}()$
  $h \leftarrow \mathsf{H}(h, c)$
  $buf.\mathsf{add}(c)$
  $pl.\mathsf{flush}()$
  $prekey_{\mathcal{J}}, prekey_{\mathcal{R}} \leftarrow \mathsf{HashObject.finalize}(ck, kk_{\mathcal{J}})$
  $k_{\mathcal{J}} \leftarrow \mathsf{AEAD.gen}(prekey_{\mathcal{J}})$
  $k_{\mathcal{R}} \leftarrow \mathsf{AEAD.gen}(prekey_{\mathcal{R}})$
  $n \leftarrow \mathsf{AEAD.Nonce.new}()$
  $pl.\mathsf{add}(m)$
  $c \leftarrow \mathsf{AEAD.enc}(k_{\mathcal{R}}, pl, h, n)$
  $n.\mathsf{increment}()$
  $buf.\mathsf{add}(c)$
  $h \leftarrow \mathsf{H}(h, c)$
  return buf

Function $\mathsf{Recv}_{\mathcal{J}}(buf)$**:**
  $c \leftarrow buf.\mathsf{parse\_next}()$
  $pl \leftarrow \mathsf{AEAD.dec}(k_0, c, h, n)$
  $h \leftarrow \mathsf{H}(h, c)$
  $n.\mathsf{increment}()$
  $ct_{\mathcal{J}} \leftarrow pl.\mathsf{parse\_next}()$
  $kk_{\mathcal{J}} \leftarrow \mathsf{IKem.dec}(sk_{\mathcal{J}}, ct_{\mathcal{J}})$
  $prekey_{\mathcal{J}}, prekey_{\mathcal{R}} \leftarrow \mathsf{HashObject.finalize}(ck, kk_{\mathcal{J}})$
  $k_{\mathcal{J}} \leftarrow \mathsf{AEAD.gen}(prekey_{\mathcal{J}})$
  $k_{\mathcal{R}} \leftarrow \mathsf{AEAD.gen}(prekey_{\mathcal{R}})$
  $n \leftarrow \mathsf{AEAD.Nonce.new}()$
  $c \leftarrow buf.\mathsf{parse\_next}()$
  $pl \leftarrow \mathsf{AEAD.dec}(k_{\mathcal{R}}, c, h, n)$
  $h \leftarrow \mathsf{H}(h, c)$
  $n.\mathsf{increment}()$
  $m \leftarrow pl.\mathsf{parse\_next}()$
  return m

Algorithm 16: pqXK

```
Function KGen_I():
    seec_sk ← SEEC.gen_key()
    pk_I, sk_I ← IKem.gen()
Function KGen_R():
    seec_sk ← SEEC.gen_key()
    pk_R, sk_R ← RKem.gen()
    Publish(pk_R)
Function Init_I():
    h ← H("pqXK_label")
    ck ← HashObject.gen("pqXK_label")
Function Send_I(m):
    pl ← String.new()
    buf ← String.new()
    r ← SEEC.gen_rand(seec_sk)
    ct_R, kk_R ← RKem.enc(pk_R, r)
    pl.add(ct_R)
    h ← H(h, pl)
    buf.add(pl)
    pl.flush()
    prekey ← ck.input(kk_R)
    k_0 ← AEAD.gen(prekey)
    n ← AEAD.Nonce.new()
    r ← SEEC.gen_rand(seec_sk)
    pk_e, sk_e ← EKem.gen(r)
    pl.add(pk_e)
    pl.add(m)
    c ← AEAD.enc(k_0, pl, h, n)
    n.increment()
    buf.add(c)
    h ← H(h, c)
    return buf
Function Init_R():
    h ← H("pqXK_label")
    ck ← HashObject.gen("pqXK_label")
Function Recv_R(buf):
    pl ← buf.parse_next()
    h ← H(h, pl)
    ct_R ← pl.parse_next()
    kk_R ← RKem.dec(sk_R, ct_R)
    prekey ← ck.input(kk_R)
    k_0 ← AEAD.gen(prekey)
    n ← AEAD.Nonce.new()
    c ← buf.parse_next()
    pl ← AEAD.dec(k_0, c, h, n)
    h ← H(h, c)
    n.increment()
    pk_e ← pl.parse_next()
    m ← pl.parse_next()
    return m
Function Send_R(m):
    pl ← String.new()
    buf ← String.new()
    r ← SEEC.gen_rand(seec_sk)
    ct_e, kk_e ← EKem.enc(pk_e, r)
    pl.add(ct_e)
    h ← H(h, pl)
    buf.add(pl)
    pl.flush()
    prekey ← ck.input(kk_e)
    k_1 ← AEAD.gen(prekey)
    n ← AEAD.Nonce.new()
    pl.add(m)
    c ← AEAD.enc(k_1, pl, h, n)
    n.increment()
    buf.add(c)
    h ← H(h, c)
    return buf
Function Recv_I(buf):
    c ← buf.parse_next()
    pl ← AEAD.dec(k_0, c, h, n)
    h ← H(h, c)
    n.increment()
    ct_e ← pl.parse_next()
    kk_e ← EKem.dec(sk_e, ct_e)
    prekey ← ck.input(kk_e)
    k_1 ← AEAD.gen(prekey)
    n ← AEAD.Nonce.new()
    c ← buf.parse_next()
    pl ← AEAD.dec(k_0, c, h, n)
    h ← H(h, c)
    n.increment()
    m ← pl.parse_next()
    return m
Function Send_I(m):
    pl ← String.new()
    buf ← String.new()
    pl.add(pk_I)
    pl.add(m)
    c ← AEAD.enc(k_1, pl, h, n)
    n.increment()
    buf.add(c)
    h ← H(h, c)
    return buf
Function Recv_R(buf):
    c ← buf.parse_next()
    pl ← AEAD.dec(k_1, c, h, n)
    h ← H(h, c)
    n.increment()
    pk_I ← pl.parse_next()
    m ← pl.parse_next()
    return m
Function Send_R(m):
    pl ← String.new()
    buf ← String.new()
    r ← SEEC.gen_rand(seec_sk)
    ct_I, kk_I ← IKem.enc(pk_I, r)
    pl.add(ct_I)
    c ← AEAD.enc(k_1, pl, h, n)
    n.increment()
    h ← H(h, c)
    buf.add(c)
    h ← H(h, c)
    return buf
Function Recv_I(buf):
    c ← buf.parse_next()
    pl ← AEAD.dec(k_1, c, h, n)
    h ← H(h, c)
    n.increment()
    ct_I ← pl.parse_next()
    kk_I ← IKem.dec(sk_I, ct_I)
    prekey_I, prekey_R ← HashObject.finalize(ck, kk_I)
    k_I ← AEAD.gen(prekey_I)
    k_R ← AEAD.gen(prekey_R)
    n ← AEAD.Nonce.new()
    pl.add(m)
    c ← AEAD.enc(k_R, pl, h, n)
    n.increment()
    buf.add(c)
    h ← H(h, c)
    return buf
Function Recv_I(buf):
    c ← buf.parse_next()
    pl ← AEAD.dec(k_1, c, h, n)
    h ← H(h, c)
    n.increment()
    ct_I ← pl.parse_next()
    kk_I ← IKem.dec(sk_I, ct_I)
    prekey_I, prekey_R ← HashObject.finalize(ck, kk_I)
    k_I ← AEAD.gen(prekey_I)
    k_R ← AEAD.gen(prekey_R)
    n ← AEAD.Nonce.new()
    c ← buf.parse_next()
    pl ← AEAD.dec(k_R, c, h, n)
    h ← H(h, c)
    n.increment()
    m ← pl.parse_next()
    return m
```

Algorithm 17: pqXX

```
Function KGen_I():
    seec_sk ← SEEC.gen_key()
    pk_I, sk_I ← IKem.gen()
Function KGen_R():
    seec_sk ← SEEC.gen_key()
    pk_R, sk_R ← RKem.gen()
Function Init_I():
    h ← H("pqXX_label")
    ck ← HashObject.gen("pqXX_label")
Function Send_I(m):
    pl ← String.new()
    buf ← String.new()
    r ← SEEC.gen_rand(seec_sk)
    pk_e, sk_e ← EKem.gen(r)
    pl.add(pk_e)
    pl.add(m)
    buf.add(pl)
    h ← H(h, pl)
    return buf
Function Init_R():
    h ← H("pqXX_label")
    ck ← HashObject.gen("pqXX_label")
Function Recv_R(buf):
    pl ← buf.parse_next()
    h ← H(h, pl)
    pk_e ← pl.parse_next()
    m ← pl.parse_next()
    return m
Function Send_R(m):
    pl ← String.new()
    buf ← String.new()
    r ← SEEC.gen_rand(seec_sk)
    ct_e, kk_e ← EKem.enc(pk_e, r)
    pl.add(ct_e)
    h ← H(h, pl)
    buf.add(pl)
    pl.flush()
    prekey ← ck.input(kk_e)
    k_0 ← AEAD.gen(prekey)
    n ← AEAD.Nonce.new()
    pl.add(pk_R)
    pl.add(m)
    c ← AEAD.enc(k_0, pl, h, n)
    n.increment()
    buf.add(c)
    h ← H(h, c)
    return buf
Function Recv_I(buf):
    pl ← buf.parse_next()
    h ← H(h, pl)
    ct_e ← pl.parse_next()
    kk_e ← EKem.dec(sk_e, ct_e)
    prekey ← ck.input(kk_e)
    k_0 ← AEAD.gen(prekey)
    n ← AEAD.Nonce.new()
    c ← buf.parse_next()
    pl ← AEAD.dec(k_0, c, h, n)
    h ← H(h, c)
    n.increment()
    pk_R ← pl.parse_next()
    m ← pl.parse_next()
    return m
Function Send_I(m):
    pl ← String.new()
    buf ← String.new()
    r ← SEEC.gen_rand(seec_sk)
    ct_R, kk_R ← RKem.enc(pk_R, r)
    pl.add(ct_R)
    c ← AEAD.enc(k_0, pl, h, n)
    n.increment()
    h ← H(h, c)
    buf.add(c)
    pl.flush()
    prekey ← ck.input(kk_R)
    k_1 ← AEAD.gen(prekey)
    n ← AEAD.Nonce.new()
    pl.add(pk_I)
    pl.add(m)
    c ← AEAD.enc(k_1, pl, h, n)
    n.increment()
    buf.add(c)
    h ← H(h, c)
    return buf
Function Recv_R(buf):
    c ← buf.parse_next()
    pl ← AEAD.dec(k_0, c, h, n)
    h ← H(h, c)
    n.increment()
    ct_R ← pl.parse_next()
    kk_R ← RKem.dec(sk_R, ct_R)
    prekey ← ck.input(kk_R)
    k_1 ← AEAD.gen(prekey)
    n ← AEAD.Nonce.new()
    c ← buf.parse_next()
    pl ← AEAD.dec(k_0, c, h, n)
    h ← H(h, c)
    n.increment()
    pk_I ← pl.parse_next()
    m ← pl.parse_next()
    return m
Function Send_R(m):
    pl ← String.new()
    buf ← String.new()
    r ← SEEC.gen_rand(seec_sk)
    ct_I, kk_I ← IKem.enc(pk_I, r)
    pl.add(ct_I)
    c ← AEAD.enc(k_1, pl, h, n)
    n.increment()
    h ← H(h, c)
    buf.add(c)
    pl.flush()
    prekey_I, prekey_R ← HashObject.finalize(ck, kk_I)
    k_I ← AEAD.gen(prekey_I)
    k_R ← AEAD.gen(prekey_R)
    n ← AEAD.Nonce.new()
    pl.add(m)
    c ← AEAD.enc(k_R, pl, h, n)
    n.increment()
    buf.add(c)
    h ← H(h, c)
    return buf
Function Recv_I(buf):
    c ← buf.parse_next()
    pl ← AEAD.dec(k_1, c, h, n)
    h ← H(h, c)
    n.increment()
    ct_I ← pl.parse_next()
    kk_I ← IKem.dec(sk_I, ct_I)
    prekey_I, prekey_R ← HashObject.finalize(ck, kk_I)
    k_I ← AEAD.gen(prekey_I)
    k_R ← AEAD.gen(prekey_R)
    n ← AEAD.Nonce.new()
    c ← buf.parse_next()
    pl ← AEAD.dec(k_R, c, h, n)
    h ← H(h, c)
    n.increment()
    m ← pl.parse_next()
    return m
```

Algorithm 18: pqIN

```
Function KGen_J():
    seec_sk ← SEEC.gen_key()
    pk_J, sk_J ← IKem.gen()
Function KGen_R():
    seec_sk ← SEEC.gen_key()
Function Init_J():
    h ← H("pqIN_label")
    ck ← HashObject.gen("pqIN_label")
Function Send_J(m):
    pl ← String.new()
    buf ← String.new()
    r ← SEEC.gen_rand(seec_sk)
    pk_e, sk_e ← EKem.gen(r)
    pl.add(pk_e)
    pl.add(pk_J)
    pl.add(m)
    buf.add(pl)
    h ← H(h, pl)
    return buf
Function Init_R():
    h ← H("pqIN_label")
    ck ← HashObject.gen("pqIN_label")
Function Recv_R(buf):
    pl ← buf.parse_next()
    h ← H(h, pl)
    pk_e ← pl.parse_next()
    pk_J ← pl.parse_next()
    m ← pl.parse_next()
    return m
Function Send_R(m):
    pl ← String.new()
    buf ← String.new()
    r ← SEEC.gen_rand(seec_sk)
    ct_e, kk_e ← EKem.enc(pk_e, r)
    pl.add(ct_e)
    h ← H(h, pl)
    buf.add(pl)
    pl.flush()
    prekey ← ck.input(kk_e)
    k_0 ← AEAD.gen(prekey)
    n ← AEAD.Nonce.new()
    r ← SEEC.gen_rand(seec_sk)
    ct_J, kk_J ← IKem.enc(pk_J, r)
    pl.add(ct_J)
    c ← AEAD.enc(k_0, pl, h, n)
    n.increment()
    h ← H(h, c)
    buf.add(c)
    pl.flush()
    prekey_J, prekey_R ← HashObject.finalize(ck, kk_J)
    k_J ← AEAD.gen(prekey_J)
    k_R ← AEAD.gen(prekey_R)
    n ← AEAD.Nonce.new()
    pl.add(m)
    c ← AEAD.enc(k_R, pl, h, n)
    n.increment()
    buf.add(c)
    h ← H(h, c)
    return buf
Function Recv_J(buf):
    pl ← buf.parse_next()
    h ← H(h, pl)
    ct_e ← pl.parse_next()
    kk_e ← EKem.dec(sk_e, ct_e)
    prekey ← ck.input(kk_e)
    k_0 ← AEAD.gen(prekey)
    n ← AEAD.Nonce.new()
    c ← buf.parse_next()
    pl ← AEAD.dec(k_0, c, h, n)
    h ← H(h, c)
    n.increment()
    ct_J ← pl.parse_next()
    kk_J ← IKem.dec(sk_J, ct_J)
    prekey_J, prekey_R ← HashObject.finalize(ck, kk_J)
    k_J ← AEAD.gen(prekey_J)
    k_R ← AEAD.gen(prekey_R)
    n ← AEAD.Nonce.new()
    c ← buf.parse_next()
    pl ← AEAD.dec(k_R, c, h, n)
    h ← H(h, c)
    n.increment()
    m ← pl.parse_next()
    return m
```

Algorithm 19: pqIK

```
1  Function KGen_I():
2      seec_sk ← SEEC.gen_key()
3      pk_I, sk_I ← IKem.gen()
4  Function KGen_R():
5      seec_sk ← SEEC.gen_key()
6      pk_R, sk_R ← RKem.gen()
7      Publish(pk_R)
8  Function Init_I():
9      h ← H("pqIK_label")
10     ck ← HashObject.gen(
         "pqIK_label")
11 Function Send_I(m):
12     pl ← String.new()
13     buf ← String.new()
14     r ← SEEC.gen_rand(seec_sk)
15     ct_R, kk_R ← RKem.enc(pk_R, r)
16     pl.add(ct_R)
17     h ← H(h, pl)
18     buf.add(pl)
19     pl.flush()
20     prekey ← ck.input(kk_R)
21     k_0 ← AEAD.gen(prekey)
22     n ← AEAD.Nonce.new()
23     r ← SEEC.gen_rand(seec_sk)
24     pk_e, sk_e ← EKem.gen(r)
25     pl.add(pk_e)
26     pl.add(pk_I)
27     pl.add(m)
28     c ← AEAD.enc(k_0, pl, h, n)
29     n.increment()
30     buf.add(c)
31     h ← H(h, c)
32     return buf
33 Function Init_R():
34     h ← H("pqIK_label")
35     ck ← HashObject.gen(
         "pqIK_label")
36 Function Recv_R(buf):
37     pl ← buf.parse_next()
38     h ← H(h, pl)
39     ct_R ← pl.parse_next()
40     kk_R ← RKem.dec(sk_R, ct_R)
41     prekey ← ck.input(kk_R)
42     k_0 ← AEAD.gen(prekey)
43     n ← AEAD.Nonce.new()
44     c ← buf.parse_next()
45     pl ← AEAD.dec(k_0, c, h, n)
46     h ← H(h, c)
47     n.increment()
48     pk_e ← pl.parse_next()
49     pk_I ← pl.parse_next()
50     m ← pl.parse_next()
51     return m
52 Function Send_R(m):
53     pl ← String.new()
54     buf ← String.new()
55     r ← SEEC.gen_rand(seec_sk)
56     ct_e, kk_e ← EKem.enc(pk_e, r)
57     pl.add(ct_e)
58     h ← H(h, pl)
59     buf.add(pl)
60     pl.flush()
61     prekey ← ck.input(kk_e)
62     k_1 ← AEAD.gen(prekey)
63     n ← AEAD.Nonce.new()
64     r ← SEEC.gen_rand(seec_sk)
65     ct_I, kk_I ← IKem.enc(pk_I, r)
66     pl.add(ct_I)
67     c ← AEAD.enc(k_1, pl, h, n)
68     n.increment()
69     h ← H(h, c)
70     buf.add(c)
71     pl.flush()
72     prekey_I, prekey_R ←
         HashObject.finalize(ck, kk_I)
73     k_I ← AEAD.gen(prekey_I)
74     k_R ← AEAD.gen(prekey_R)
75     n ← AEAD.Nonce.new()
76     pl.add(m)
77     c ← AEAD.enc(k_R, pl, h, n)
78     n.increment()
79     buf.add(c)
80     h ← H(h, c)
81     return buf
82 Function Recv_I(buf):
83     c ← buf.parse_next()
84     pl ← AEAD.dec(k_0, c, h, n)
85     h ← H(h, c)
86     n.increment()
87     ct_e ← pl.parse_next()
88     kk_e ← EKem.dec(sk_e, ct_e)
89     prekey ← ck.input(kk_e)
90     k_1 ← AEAD.gen(prekey)
91     n ← AEAD.Nonce.new()
92     c ← buf.parse_next()
93     pl ← AEAD.dec(k_0, c, h, n)
94     h ← H(h, c)
95     n.increment()
96     ct_I ← pl.parse_next()
97     kk_I ← IKem.dec(sk_I, ct_I)
98     prekey_I, prekey_R ←
         HashObject.finalize(ck, kk_I)
99     k_I ← AEAD.gen(prekey_I)
100    k_R ← AEAD.gen(prekey_R)
101    n ← AEAD.Nonce.new()
102    c ← buf.parse_next()
103    pl ← AEAD.dec(k_R, c, h, n)
104    h ← H(h, c)
105    n.increment()
106    m ← pl.parse_next()
107    return m
```

Algorithm 20: pqIX

```
Function KGen_J():
    seec_sk ← SEEC.gen_key()
    pk_J, sk_J ← IKem.gen()
Function KGen_R():
    seec_sk ← SEEC.gen_key()
    pk_R, sk_R ← RKem.gen()
Function Init_J():
    h ← H("pqIX_label")
    ck ← HashObject.gen("pqIX_label")
Function Send_J(m):
    pl ← String.new()
    buf ← String.new()
    r ← SEEC.gen_rand(seec_sk)
    pk_e, sk_e ← EKem.gen(r)
    pl.add(pk_e)
    pl.add(pk_J)
    pl.add(m)
    buf.add(pl)
    h ← H(h, pl)
    return buf
Function Init_R():
    h ← H("pqIX_label")
    ck ← HashObject.gen("pqIX_label")
Function Recv_R(buf):
    pl ← buf.parse_next()
    h ← H(h, pl)
    pk_e ← pl.parse_next()
    pk_J ← pl.parse_next()
    m ← pl.parse_next()
    return m
Function Send_R(m):
    pl ← String.new()
    buf ← String.new()
    r ← SEEC.gen_rand(seec_sk)
    ct_e, kk_e ← EKem.enc(pk_e, r)
    pl.add(ct_e)
    h ← H(h, pl)
    buf.add(pl)
    pl.flush()
    prekey ← ck.input(kk_e)
    k_0 ← AEAD.gen(prekey)
    n ← AEAD.Nonce.new()
    r ← SEEC.gen_rand(seec_sk)
    ct_J, kk_J ← IKem.enc(pk_J, r)
    pl.add(ct_J)
    c ← AEAD.enc(k_0, pl, h, n)
    n.increment()
    h ← H(h, c)
    buf.add(c)
    pl.flush()
    prekey ← ck.input(kk_J)
    k_1 ← AEAD.gen(prekey)
    n ← AEAD.Nonce.new()
    pl.add(pk_R)
    pl.add(m)
    c ← AEAD.enc(k_1, pl, h, n)
    n.increment()
    buf.add(c)
    h ← H(h, c)
    return buf
Function Recv_J(buf):
    pl ← buf.parse_next()
    h ← H(h, pl)
    ct_e ← pl.parse_next()
    kk_e ← EKem.dec(sk_e, ct_e)
    prekey ← ck.input(kk_e)
    k_0 ← AEAD.gen(prekey)
    n ← AEAD.Nonce.new()
    c ← buf.parse_next()
    pl ← AEAD.dec(k_0, c, h, n)
    h ← H(h, c)
    n.increment()
    ct_J ← pl.parse_next()
    kk_J ← IKem.dec(sk_J, ct_J)
    prekey ← ck.input(kk_J)
    k_1 ← AEAD.gen(prekey)
    n ← AEAD.Nonce.new()
    c ← buf.parse_next()
    pl ← AEAD.dec(k_0, c, h, n)
    h ← H(h, c)
    n.increment()
    pk_R ← pl.parse_next()
    m ← pl.parse_next()
    return m
Function Send_J(m):
    pl ← String.new()
    buf ← String.new()
    r ← SEEC.gen_rand(seec_sk)
    ct_R, kk_R ← RKem.enc(pk_R, r)
    pl.add(ct_R)
    c ← AEAD.enc(k_1, pl, h, n)
    n.increment()
    h ← H(h, c)
    buf.add(c)
    pl.flush()
    prekey_J, prekey_R ← HashObject.finalize(ck, kk_R)
    k_J ← AEAD.gen(prekey_J)
    k_R ← AEAD.gen(prekey_R)
    n ← AEAD.Nonce.new()
    pl.add(m)
    c ← AEAD.enc(k_J, pl, h, n)
    n.increment()
    buf.add(c)
    h ← H(h, c)
    return buf
Function Recv_R(buf):
    c ← buf.parse_next()
    pl ← AEAD.dec(k_1, c, h, n)
    h ← H(h, c)
    n.increment()
    ct_R ← pl.parse_next()
    kk_R ← RKem.dec(sk_R, ct_R)
    prekey_J, prekey_R ← HashObject.finalize(ck, kk_R)
    k_J ← AEAD.gen(prekey_J)
    k_R ← AEAD.gen(prekey_R)
    n ← AEAD.Nonce.new()
    c ← buf.parse_next()
    pl ← AEAD.dec(k_J, c, h, n)
    h ← H(h, c)
    n.increment()
    m ← pl.parse_next()
    return m
```

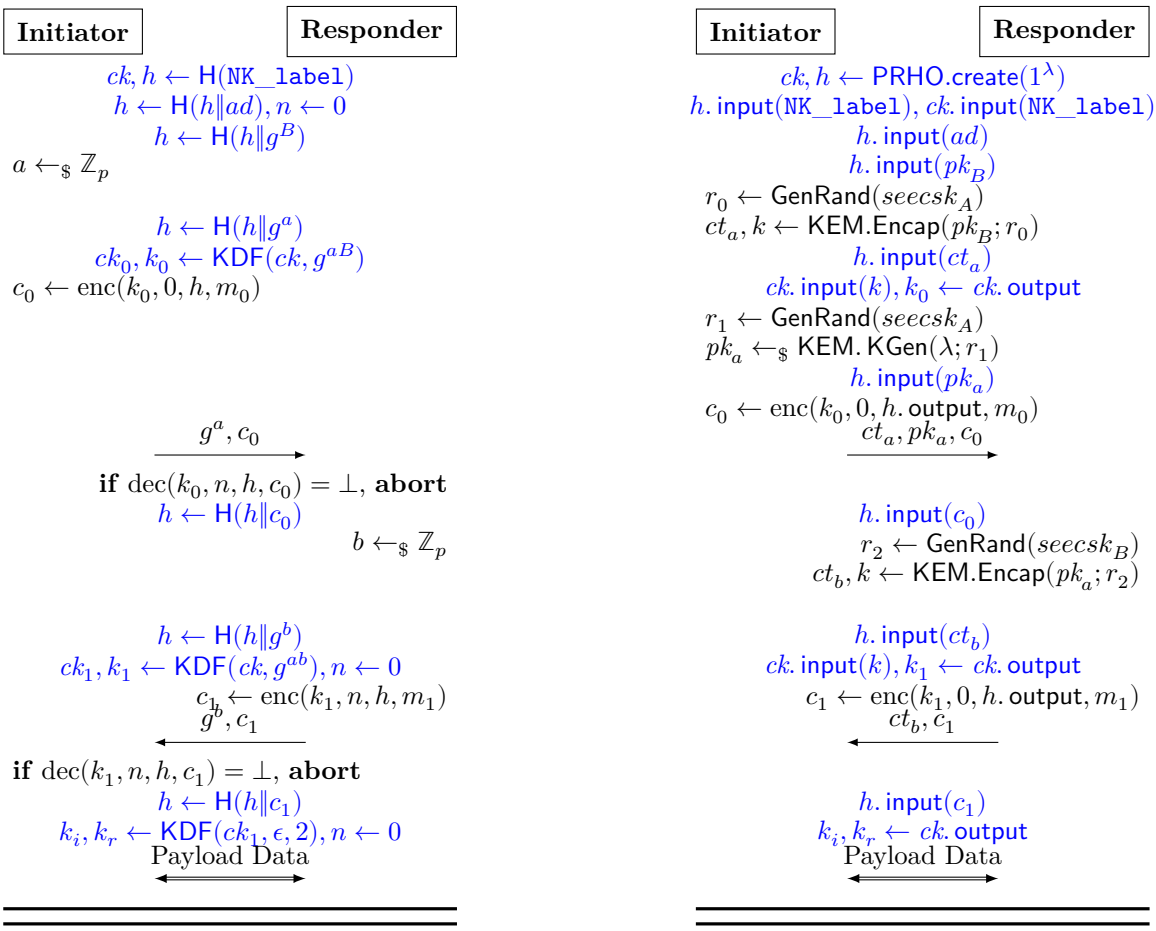

Figure 6: The NK patterns of classical Noise (left) and PQNoise (right). For reasons of space we use the following conventions here: highlighted actions are performed by both parties as soon as they receive all necessary values to perform the computations in question. If any decryption- or decapsulation-algorithm returns $\perp$, the party in question aborts the protocol.

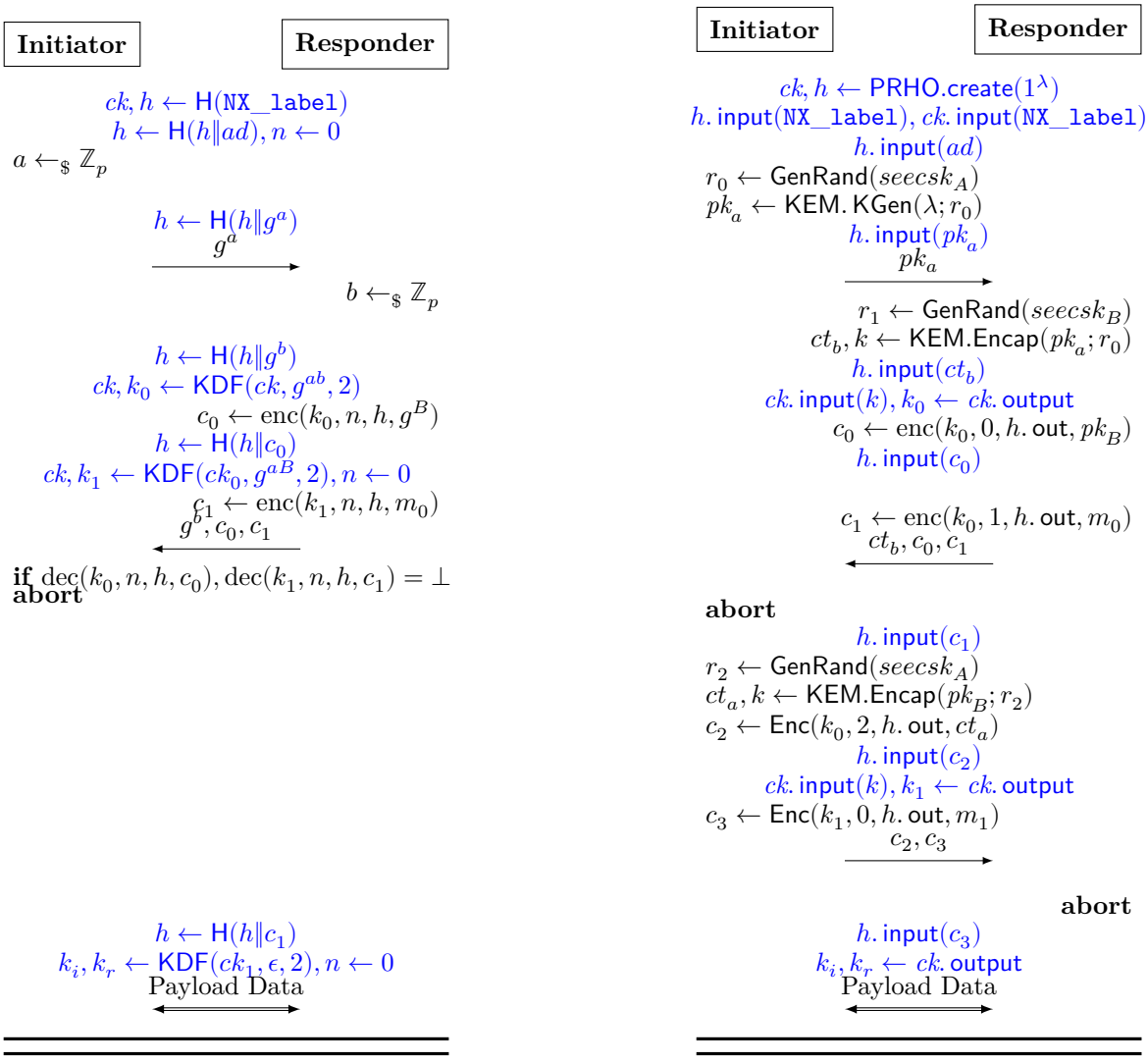

Figure 7: The NX patterns of classical Noise (left) and PQNoise (right). For reasons of space we use the following conventions here: highlighted actions are performed by both parties as soon as they receive all necessary values to perform the computations in question. If any decryption- or decapsulation-algorithm returns $\perp$, the party in question aborts the protocol.

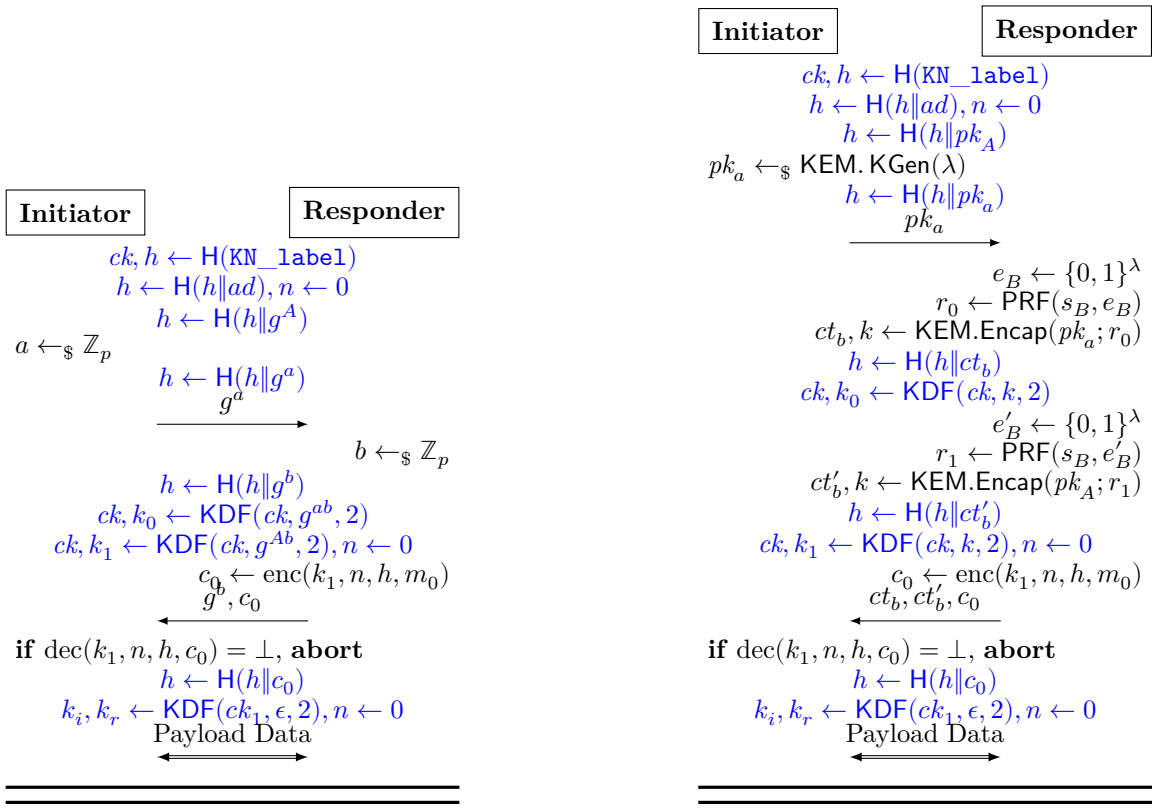

Figure 8: The KN patterns of classical Noise (left) and PQNoise (right). For reasons of space we use the following conventions here: highlighted actions are performed by both parties as soon as they receive all necessary values to perform the computations in question. If any decryption- or decapsulation-algorithm returns $\perp$, the party in question aborts the protocol.

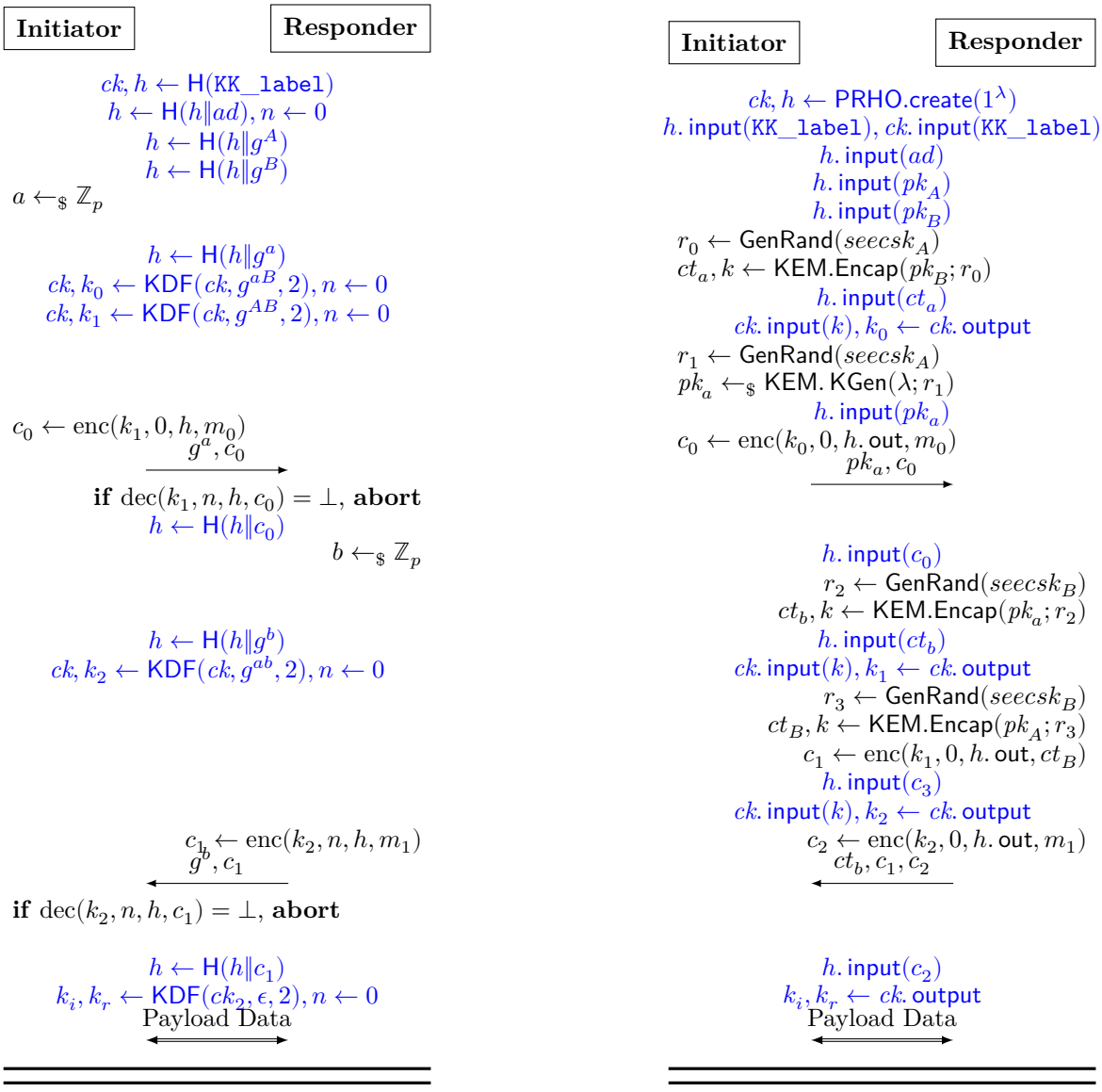

Figure 9: The KK patterns of classical Noise (left) and PQNoise (right). For reasons of space we use the following conventions here: highlighted actions are performed by both parties as soon as they receive all necessary values to perform the computations in question. If any decryption- or decapsulation-algorithm returns $\perp$, the party in question aborts the protocol.

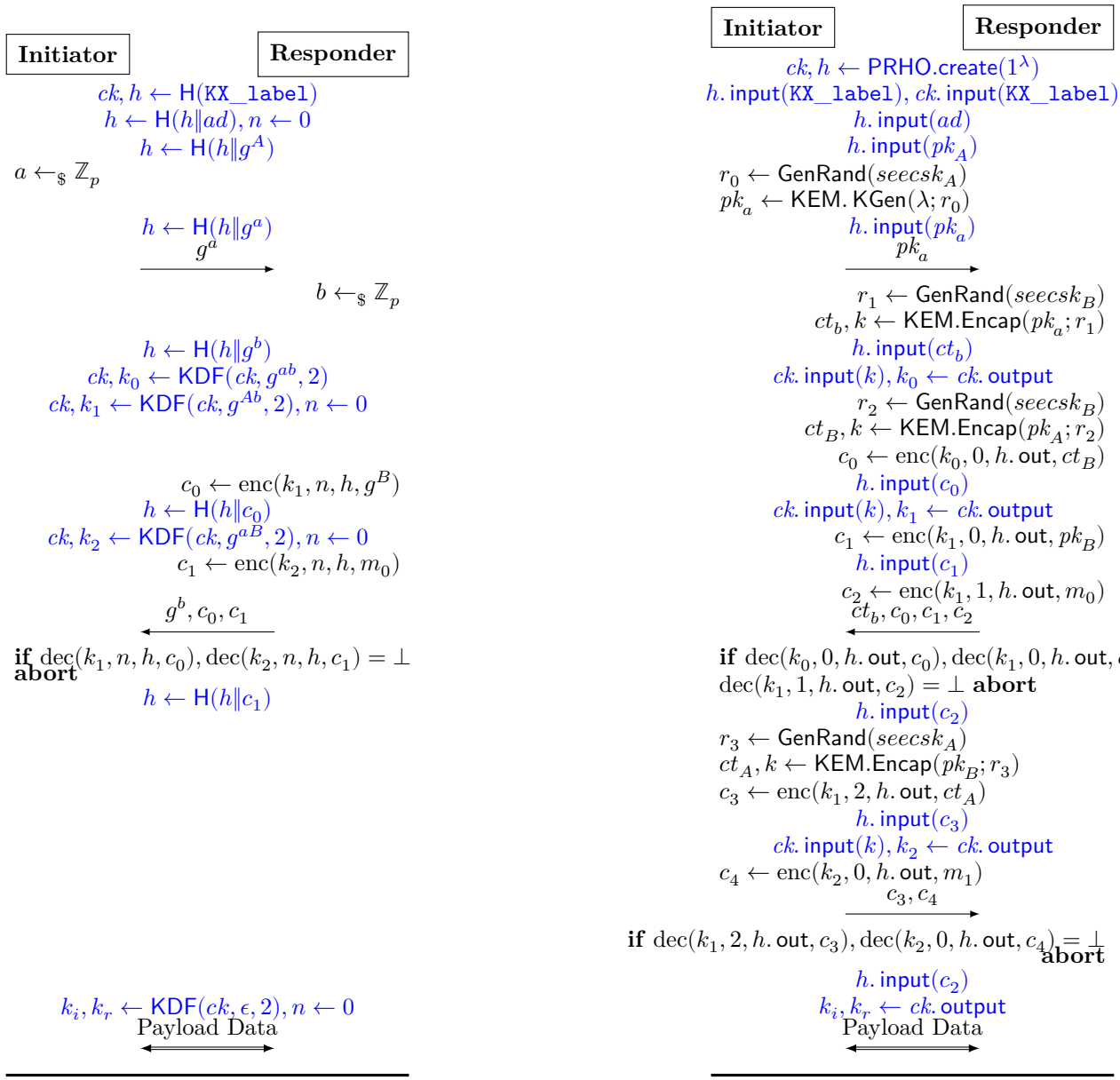

Figure 10: The KX patterns of classical Noise (left) and PQNoise (right). For reasons of space we use the following conventions here: highlighted actions are performed by both parties as soon as they receive all necessary values to perform the computations in question. If any decryption- or decapsulation-algorithm returns $\perp$, the party in question aborts the protocol.

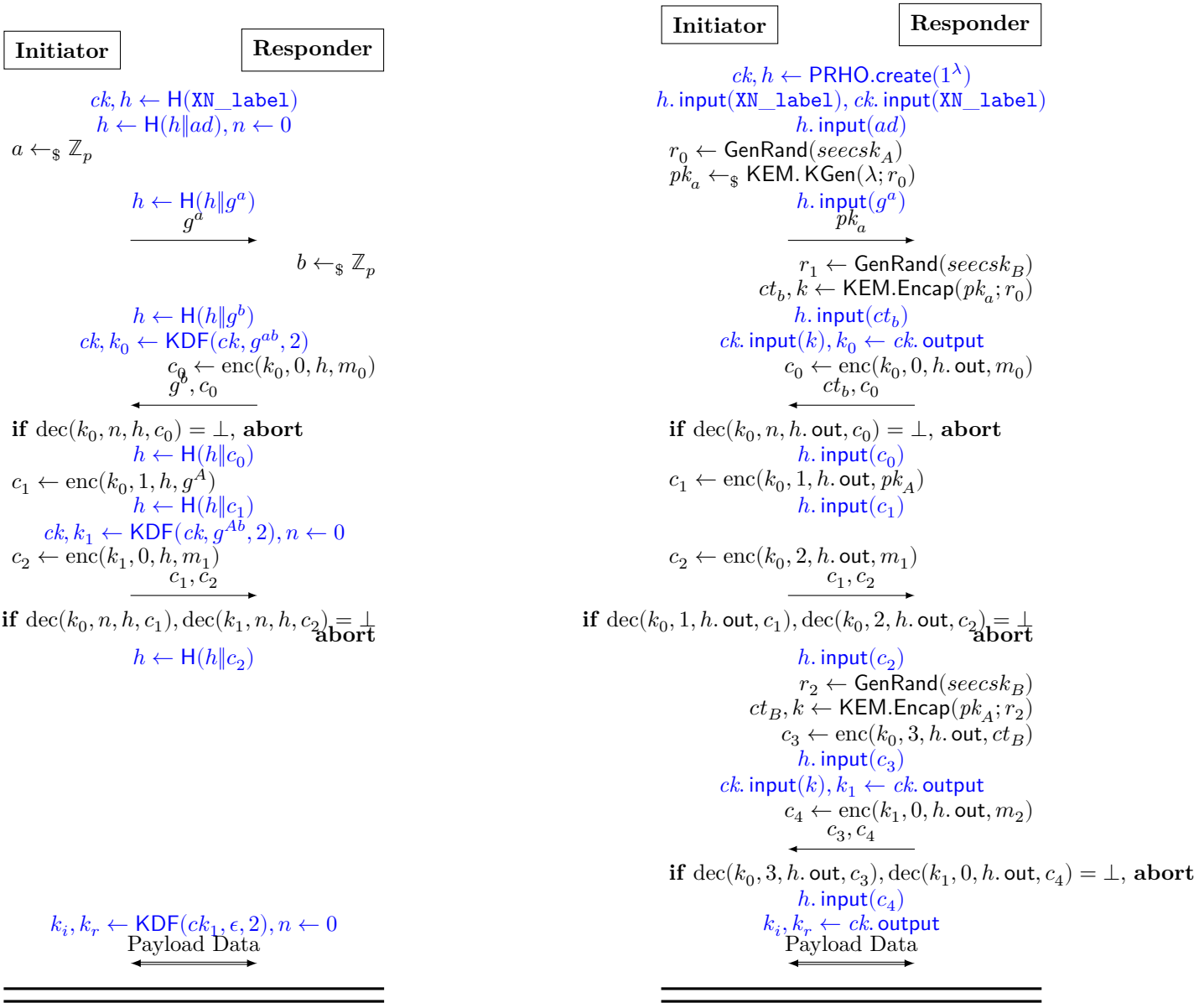

Figure 11: The XN patterns of classical Noise (left) and PQNoise (right). For reasons of space we use the following conventions here: highlighted actions are performed by both parties as soon as they receive all necessary values to perform the computations in question. If any decryption- or decapsulation-algorithm returns $\perp$, the party in question aborts the protocol.

Figure 12: The XK 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.

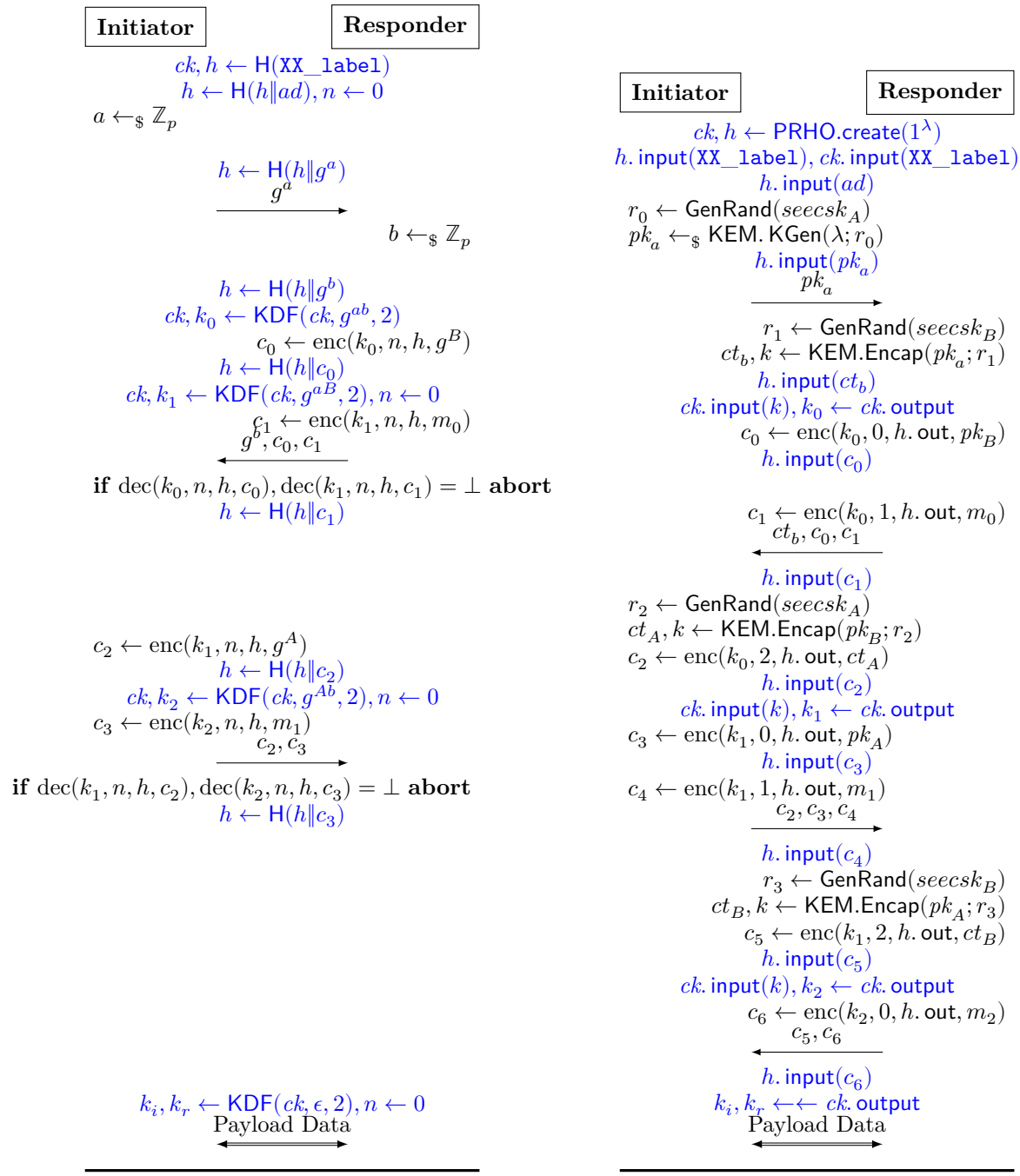

Figure 13: The XX patterns of classical Noise (left) and PQNoise (right). For reasons of space we use the following conventions here: highlighted actions are performed by both parties as soon as they receive all necessary values to perform the computations in question. If any decryption- or decapsulation-algorithm returns $\perp$, the party in question aborts the protocol.

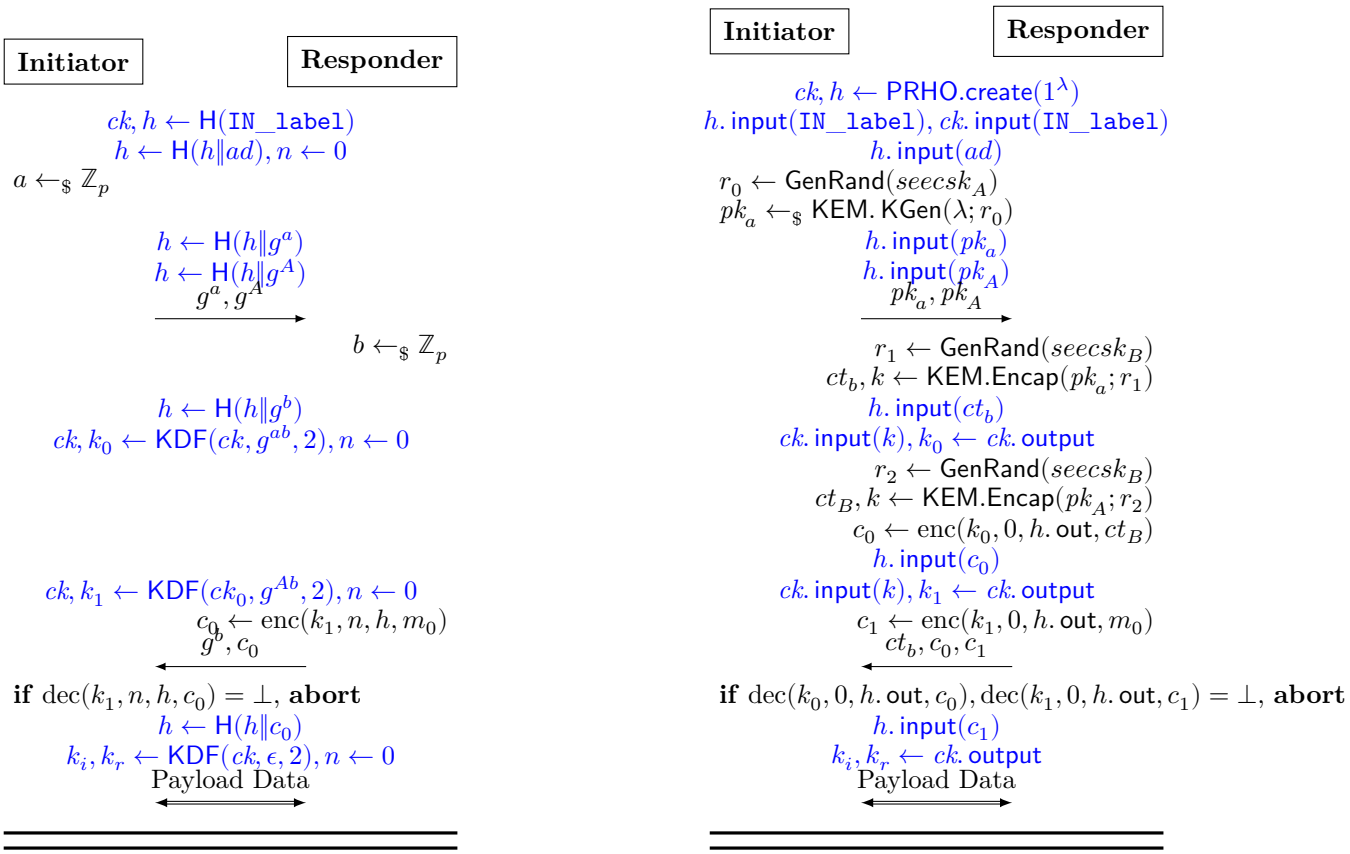

Figure 14: The IN patterns of classical Noise (left) and PQNoise (right). For reasons of space we use the following conventions here: highlighted actions are performed by both parties as soon as they receive all necessary values to perform the computations in question. If any decryption- or decapsulation-algorithm returns $\perp$, the party in question aborts the protocol.

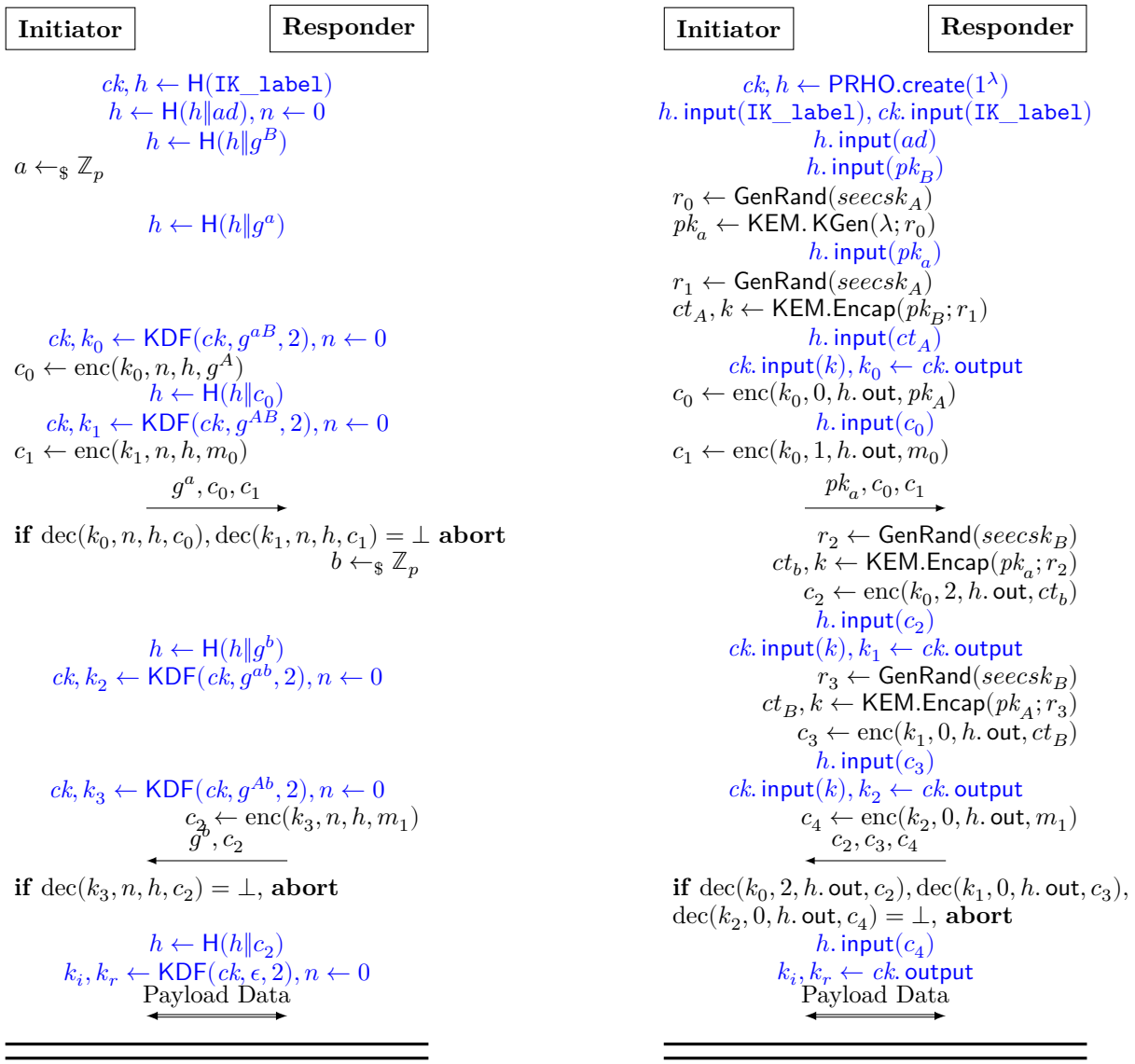

Figure 15: The IK patterns of classical Noise (left) and PQNoise (right). For reasons of space we use the following conventions here: highlighted actions are performed by both parties as soon as they receive all necessary values to perform the computations in question. If any decryption- or decapsulation-algorithm returns $\perp$, the party in question aborts the protocol.

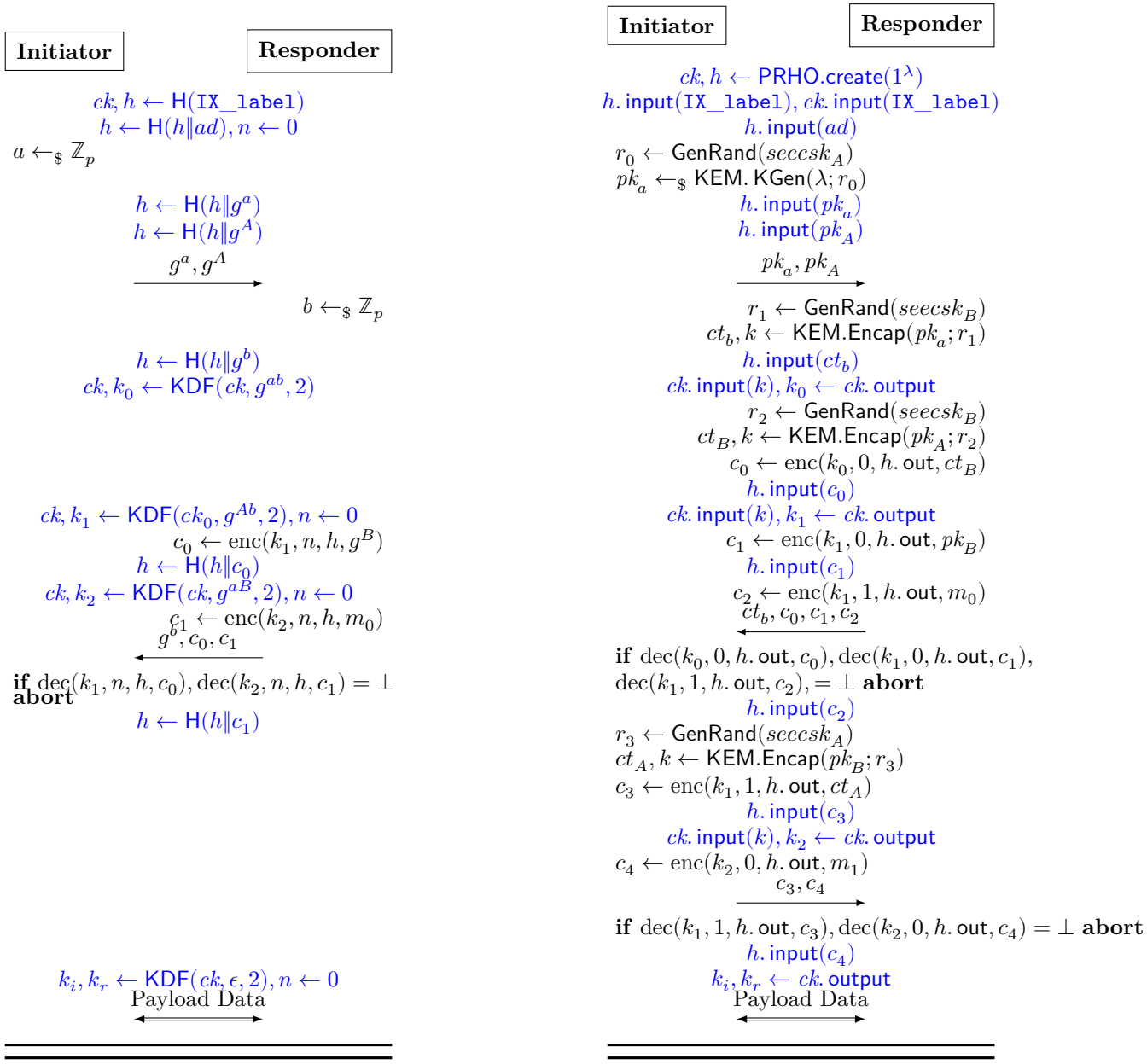

Figure 16: The IX patterns of classical Noise (left) and PQNoise (right). For reasons of space we use the following conventions here: highlighted actions are performed by both parties as soon as they receive all necessary values to perform the computations in question. If any decryption- or decapsulation-algorithm returns $\perp$, the party in question aborts the protocol.