A Formal Analysis of Apple's iMessage PQ3 Protocol

Felix Linker
Department of Computer Science, ETH Zurich

Ralf Sasse
Department of Computer Science, ETH Zurich

David Basin
Department of Computer Science, ETH Zurich

Abstract

We present the formal verification of Apple's iMessage PQ3, a highly performant, device-to-device messaging protocol offering strong security guarantees even against an adversary with quantum computing capabilities. PQ3 leverages Apple's identity services together with a custom, post-quantum secure initialization phase and afterwards it employs a double ratchet construction in the style of Signal, extended to provide post-quantum, post-compromise security.

We present a detailed formal model of PQ3, a precise specification of its fine-grained security properties, and machine-checked security proofs using the TAMARIN prover. Particularly novel is the integration of post-quantum secure key encapsulation into the relevant protocol phases and the detailed security claims along with their complete formal analysis. Our analysis covers both key ratchets, including unbounded loops, which was believed by some to be out of scope of symbolic provers like TAMARIN (it is not!).

1 Introduction

Research on secure instant messaging goes back over two decades, with early proposals including Off-the-Record Messaging [1], the Silent Circle Instant Messaging Protocol [2], iMessage, and Signal [3, 4, 5]. Over time, the security community's understanding of the threat models and security claims for secure messaging evolved. Modern messaging protocols now offer strong guarantees and can communicate messages secretly even in the presence of adversaries who corrupt different parties in different ways during the protocol's execution. This is befitting given that strong adversaries, like nation states, are capable of compromising both messaging servers and the end points sending and receiving messages. More recently, security against adversaries with quantum computing capabilities has also become an important concern. This requires protection against adversaries who can "harvest now and decrypt later," namely adversaries who leverage the decreasing cost of mass storage to store the encrypted data they intercept and to decrypt it in the future when quantum computers become sufficiently powerful [6].

In this paper, we present our formal analysis of Apple's advanced, widely deployed iMessage PQ3 Messaging Protocol, or PQ3 for short. PQ3 is used across all of Apple's devices for device-to-device messaging and underlies many other Apple services, e.g., iMessage, FaceTime, HomeKit, and HomePod hand-off. PQ3 is designed to be performant and to offer strong guarantees against powerful adversaries, including those who later possess quantum computers.

PQ3 employs a double-ratchet construction similar to Signal [3]. The protocol takes a hybrid approach to security and combines classical cryptographic primitives, like elliptic curve Diffie-Hellman, and post-quantum primitives, namely ML-KEM [7], a module-lattice-based key-encapsulation mechanism. The hybrid construction means that PQ3's security does not solely depend on the security of post-quantum primitives, which are less well understood than their classic counterparts. Moreover, PQ3's integration of hybrid cryptography into the double ratchet provides stronger guarantees than Signal, where a post-quantum Key Encapsulation Mechanism (KEM) is just integrated into the protocol's setup phase, but not into its ratcheting (see Section 2).

We analyzed PQ3's security in detail using the TAMARIN prover [8, 9], a state-of-the-art security protocol model checker. Our formal models and proofs are accessible on GitHub [10]. We report on our model of PQ3, the adversary assumptions, and the protocol's desired properties. We use TAMARIN's specification language to specify the messaging protocol and its use of classical and post-quantum cryptography. We also specify all forms of adversary compromise, including the event in which the attacker obtains a sufficiently powerful quantum computer, allowing them to break all non-post-quantum-secure cryptographic primitives. Essentially, the adversary can compromise any key at any time, either through dedicated key-reveal rules or because they obtained a quantum computer. Using TAMARIN's property language, we formalize and prove both secrecy and authenticity theorems. These theorems precisely express the protocol's security guarantees capturing fine-grained notions of key compromise.

Our analysis establishes that PQ3 provides strong security guarantees against an active network adversary that can compromise any secret key, unless explicitly stated otherwise. For example, PQ3 provides forward secrecy, post-compromise security, and post-quantum security with respect to a "harvest now, decrypt later" adversary. In contrast to Signal, PQ3 provides post-compromise security also against active classical and "harvest now, decrypt later" adversaries and not only against passive, classical adversaries. Moreover, the fine-grained analysis of compromise possibilities and their effects is useful for guiding secure implementations of PQ3. For example, the compromise of a participant's long-term identity key impacts all security guarantees and thus should be stored with extra care, for example, in a device's secure enclave.

Contributions Our first contribution is the formalization and machine-checked verification of PQ3 to prove all our security claims. Namely, we use TAMARIN to prove that PQ3 offers strong security guarantees against a powerful adversary with quantum computing capabilities. These guarantees are both fine-grained and tight in that omitting any of the many adversary compromise cases leads to attacks. Our verification thereby provides a formal, machine-checked proof that PQ3 meets the high expectations for a modern device-to-device messaging protocol. This high assurance is important given the prominent role of this protocol, which is used in billions of devices worldwide, and its limited prior analysis.

Our second contribution is to show that symbolic security protocol model checkers, in particular TAMARIN, can verify substantial, real-world protocols with nested loops, in their full complexity. This is non-trivial as it entails reasoning about unboundedly many parallel instances of the protocol, where the runs (two devices sending messages) are themselves unbounded. In fact, it was commonly believed that "unbounded (looping) protocols like Signal, and protocols with mutable recursive data structures [...] are also out of scope for symbolic provers, without introducing artificial restrictions" [11]. Our work shows that this is not the case and provides a general methodology for carrying out such proofs.

Organization In Section 2 we survey related work on messaging protocols and their verification. Afterwards, in Sections 3 and 4 we describe PQ3's threat model, requirements, and the protocol itself. In Section 5 we present our TAMARIN model of PQ3, the adversary, the protocol's properties, and details on our proofs. We draw conclusions in Section 6.

2 Related work

2.1 Messaging Protocols

Over the past decades, hundreds of secure messaging systems have been proposed [12]. The underlying protocols differ in how they bootstrap trust to set up initial keys, the properties they achieve, the adversaries they consider, whether bilateral or group communication is supported, and usability. The strongest protocols support message secrecy and authenticity against very strong adversaries. As servers cannot be trusted, encryption must be carried out end-to-end. Moreover, it is common to consider adversaries who can compromise agents' long-term secrets, and even their session states.

The security bar is now quite high. Modern protocols like Signal, which is used for example in the Signal app, WhatsApp, and Facebook Secret Conversations, offer security guarantees, even when adversaries compromise the devices of the agents running the protocol. In particular, Signal supports both forward secrecy and post-compromise security [13, 14] (also called self-healing or backward secrecy). The former protects the protocol's participants against the future compromise of past sessions, for example, the loss of a long-term secret should not jeopardize the secrecy of previously exchanged messages. The latter helps the participants to recover or "self-heal" from a past compromise to communicate secretly again in the present and future.

Messaging protocols achieve these strong properties by using *ratcheting*, an approach to continually generate new keys. Ratcheting was first proposed in the Off-the-Record Messaging [1] protocol where, with each message round trip, users establish a fresh ephemeral Diffie-Hellman shared secret. Signal further developed this idea with their *double-ratchet algorithm* [3], which nests two ratchets: an outer public-key ratchet and an inner symmetric-key ratchet. This mechanism ensures that the symmetric keys used for encryption and decryption are updated with every message sent, as opposed to just on every round trip. The protocol can recover from past compromises on every round-trip due to a new Diffie-Hellman secret. Forward secrecy is achieved for the symmetric keys as the ratchet chain does not allow one to compute the previous keys from the current message encryption key, but only the future ones.

More recently, researchers have investigated improvements offering guarantees against adversaries with quantum computing capabilities. The Signal protocol uses the Extended Triple Diffie-Hellman (X3DH) Key Agreement Protocol [5] to negotiate the session key used as the ratchet's initial root key. The recently developed PQXDH Key Agreement Protocol [4] strengthens X3DH by additionally incorporating a post-quantum KEM like Crystals-Kyber [15], and has been verified using both ProVerif and CryptoVerif [16], as well as with a pen-and-paper game-based reduction proof [17]. It has been proven (see Section 2.2) that PQXDH provides forward secrecy even in the presence of an adversary with quantum computing capabilities, provided all KEM private keys remain uncompromised. However, as the post-quantum KEM is only used in the setup phase, the subsequent use of Signal's double ratchet does not provide post-compromise security against an adversary with quantum computing capabilities, which PQ3 does. Note that both X3DH and PQXDH additionally provide cryptographic deniability [18], which is not provided by PQ3 and hence out of scope for our work.

2.2 Verification of Messaging Protocols

There has been considerable research on verifying messaging protocols using sophisticated constructions like the double ratchet to achieve strong security guarantees. Researchers have studied Signal and variants of it from both a computational and a symbolic perspective, using both pen-and-paper and machine-checked proofs.

Computational proofs A number of pen-and-paper proofs of messaging protocols involving double ratchets have been constructed in the computational setting. This means, in contrast to the symbolic model (introduced shortly), that protocols are analyzed with respect to computational definitions of security. Agents manipulate bit strings, the adversary's capabilities are modeled by probabilistic polynomial-time Turing machines, and security definitions are thus probabilistic. These models support a more detailed analysis of cryptography than symbolic abstractions. However, the proofs can be quite complex and hence they typically involve their own abstractions or protocol simplifications. Moreover, given that the proofs are traditional pen-and-paper arguments, they are more error-prone than proofs checked by computers. There are exceptions, namely computational proofs constructed with tools like CryptoVerif [19], but these are usually limited to the study of relatively simple combinations of primitives, not complex protocols like the full Signal or PQ3 double ratchet.

In [20], the authors analyze variants of the double ratchet protocol in the Universal Composability framework. As part of their analysis, they consider when keys must be deleted for different properties to hold. Their proofs are game-based with detailed security definitions. Game-based proofs are also given by [21, 22, 23]. In particular, [22] presents a formal analysis of Signal in the random oracle model. Their focus is on Signal's key agreement and they reason about loops using induction. [23] carries out game-based proofs for a Signal-like protocol; they provide a rational reconstruction of a generalized protocol that modularly achieves the different kinds of properties one wants from Signal and the use of double ratchets. In all these works, security is shown using pen-and-paper proofs, which are not machine checked, and post-quantum security is not considered.

Concomitantly to our work, Stebila carried out a computational analysis of PQ3 [24], providing a reduction argument for its security. He also formalizes the hybrid cryptography integrated into both PQ3's initialization and double ratchet, and establishes that this provides both forward secrecy and post-compromise security against both classical and "harvest now, decrypt later" adversaries. The modeling of cryptography is, as is standard for computational formalizations, more concrete and detailed than in our approach. In contrast, the security model, and the proofs (which are game-based, focused on deriving a bound on the adversary's advantages) are considerably more complex, and proofs are pen-and-paper based, rather than machine checked.

We believe, as Apple researchers also do, that there is substantial benefit to having both kinds of proofs, as they both have their relative strengths. Computational proofs capture the detailed cryptographic assumptions on the operators used. They can also capture the adversary's advantage in attacking a protocol, by bounding the probability of success for an adversary with given computational resources. In contrast, symbolic proofs better support machine-checked proofs, using different computer-supported proof techniques, like constraint solving and mathematical induction. This supports giving detailed models of protocols' and adversary's operational semantics, considering unboundedly many protocol participants and interleaved parallel sessions, and verifying these against detailed, fine-grained security properties.

This value of symbolic proofs is exemplified by our analysis of *injective agreement* [25] (Section 5.3.2), which formalizes that a protocol provides replay protection. [24] did not consider replay in its analysis, and during our TAMARIN proofs, we uncovered that injective agreement can only be provided under additional assumptions (not present in [24]) on the session-handling layer.

Symbolic proofs In terms of verification, the works closest to ours use the symbolic model of cryptography. In this model, messages are represented as terms in a term algebra (rather than bit-strings) and one uses possibilistic rather than probabilistic definitions of security. TAMARIN [8, 9] and ProVerif [26] are examples of tools constructing proofs in this setting. For example, to show that a key is a secret, one would use these tools to prove that, no matter how arbitrarily many protocol runs are interleaved, including runs where the adversary is active, the adversary cannot possibly learn the intended secret. Such proofs may be constructed automatically or interactively, and attempts to prove false statements generally yield attacks on the specified properties.

[27] analyzes Signal's session-handling layer Sesame. They use TAMARIN to show that, when sessions are accounted for, Signal does not achieve post-compromise security despite the double ratchet having this property. In this work, we do not consider PQ3's session handling layer as its specification was not made available to us. Analyzing PQ3 in conjunction with session handling is an interesting line of future work.

[28] use ProVerif and CryptoVerif [19] to analyze a variant of Signal where they extract the models they analyze from an implementation in a JavaScript dialect. Their models are substantially simplified. For example, they lack the inner ratchet based on symmetric cryptography and only consider a fixed, finite number of protocol sessions without loops.

As previously explained, Signal uses the X3DH protocol to agree on a shared key (the initial root key) prior to the double ratchet's start. The post-quantum version PQXDH has been analyzed in [16] both symbolically, using ProVerif, and computationally, using CryptoVerif. As the authors explain "Notably, this is the first machine-checked post-quantum security proof of a real-world cryptographic protocol." While this is indeed the case, they only consider the initialization part of the Signal protocol. They do not reason about the double ratchet construction, which is based on classical cryptography and thus provides no post-quantum security guarantees.

In [11], the authors analyzed Signal based on an F* implementation. They observe: "Notably, Signal has not been mechanically analyzed for an arbitrary number of rounds before. The ProVerif analysis of the Signal protocol in [28] was limited to two messages (three ratcheting rounds), at which point the analysis already took 29 hours. (With CryptoVerif, the analysis of Signal has to be limited to just one ratcheting round)." Their own proof is however also limited and only verifies properties for the outermost ratchet. In contrast, our proof uses induction within TAMARIN to machine check proofs about both ratchets of PQ3. Even in the classical setting, ignoring our post-quantum extensions, verifying the inner ratchet allows us to establish security properties against stronger adversaries who can compromise session state during the inner ratchet's execution.

3 Requirements and Threat Model

3.1 Security Requirements

Secrecy PQ3 was designed to provide strong secrecy guarantees, namely *message secrecy*, *forward secrecy*, and *post-compromise security*. Message secrecy means that as long as neither participants' session states are revealed, the adversary cannot learn any of their exchanged messages. Forward secrecy and post-compromise security limit the window in which an adversary can learn exchanged messages after they compromise parts of the session state. We discussed forward secrecy and post-compromise security already in Section 2.1. In short, forward secrecy protects protocol participants against the future compromise of past sessions, and post-compromise security helps to recover or "self-heal" from a past compromise to communicate secretly again in the present and future.

In our security analysis, we define a secrecy lemma that captures all three notions of secrecy and that addresses the precise implications of partial session state compromise. Describing this fine-grained secrecy lemma requires a detailed understanding of the key material used in PQ3, and is thus deferred to Section 5.3.1.

Authentication and Replay Protection A message recipient can identify the message's sender. We formulate this as an *agreement property*: the recipient and sender agree on their view of the message. For any message received, allegedly originating from the peer at message counter i, the peer must have actually sent the message using counter i, intending it to go to the receiver. Moreover, this agreement is *injective* [25]. Namely, a given message is only accepted once by the recipient; hence the protocol provides *replay protection*.

Note that PQ3 is a device-to-device messaging protocol and group messaging is not in PQ3's direct scope. Hence, a security analysis of group aspects (such as members joining or leaving groups) is not part of our analysis. In practice, group messaging scenarios, involving groups of devices, are handled by sending messages via pairwise runs of PQ3 to all group members, respectively to all of a user's devices. Thus, any such functionality is provided by PQ3's session-handling layer and outside of our analysis.

3.2 Threat Model

PQ3 seeks to provide the above security properties even when the protocol is run in the presence of a strong active network adversary who may have access to a powerful quantum computer in the future. As an active network adversary, the adversary can read, reorder, intercept, replay, and send any message to any participant. We assume though that devices use strong randomness and that, short of possessing a quantum computer, the adversary cannot factor large numbers or compute discrete logs. Hence, in the pre-quantum era, cryptographic primitives like (elliptic curve) Diffie-Hellman are secure against the adversary.

Our threat model explicitly accounts for the possibility that the adversary may at some point possess a quantum computer. The adversary anticipates future developments in quantum computing and stores all messages sent by the protocol participants. Therefore, they may be able to break classical cryptographic primitives later and decrypt messages, when sufficiently powerful quantum computers become available. For this reason, the adversary is referred to as a "harvest now, decrypt later" adversary.

We make two assumptions on the adversary's future quantum computing capabilities. First, the adversary will never be able to decapsulate secrets that were encapsulated with a post-quantum secure KEM like ML-KEM. They can only break other cryptographic mechanisms using a quantum computer, in particular elliptic curve Diffie-Hellman. Second, as soon as the adversary possesses a quantum computer, no honest participant runs the protocol. In other words, the adversary is only a passive quantum attacker, which models that they "harvest now, decrypt *later*." PQ3 only protects past sessions against quantum attackers. To protect active sessions, PQ3's relies on an elliptic curve signature scheme, which can be broken by a quantum computer.

We make no assumptions about the security of the devices running the protocol. Hence, the adversary can additionally compromise any key at any time and our security properties must therefore account for this. When we analyze our fine-grained security properties, we will consider the compromise of secret keys individually, e.g., the possibility that some, but not all keys, are simultaneously compromised. This reflects that some keys may be in main memory and relatively easy to compromise, whereas others (like identity keys) are stored in Apple's Secure Enclave and are thus much harder to compromise.

For setup and session establishment, the protocol leverages Apple's IDentity Services (IDS) key directory. We assume that this directory is secure in that it only distributes the participants' authentic public keys. The problem of key authentication is orthogonal to PQ3 and has recently been addressed by Apple with their rollout of "Contact Key Verification" [29].

4 PQ3 Messaging Protocol

PQ3 is a device-to-device messaging protocol where either device can asynchronously exchange messages at any time, independent of the connection status of their peer's device. We first describe PQ3 at a high-level of abstraction, followed by a more detailed account. The protocol is described in full in Appendix C.

4.1 High-level Account

In PQ3, communication between two parties, say Alice and Bob, works roughly as follows. Suppose that Alice wants to initiate messaging with Bob.

1. Alice queries Apple's IDentity Service (IDS) for Bob's *pre-key material* and a *long-term identity public key*.

2. Alice derives an initial *root key*, *chain key*, and *message key*. Alice encrypts her first message for Bob using the message key and sends Bob the ciphertext along with a signature and the key material necessary to derive the initial root key.

3. Upon receiving this new message, Bob lacks the key to decrypt the ciphertext, and so he must derive it. Bob first queries the IDS to verify Alice's long-term identity public key and checks the received signature. He uses the key material received from Alice to derive the initial root, chain, and message key and decrypts the initial message. Alice and Bob have now established a shared session.

4. As long as the session does not change direction (i.e., the current sender keeps sending messages), both parties perform *symmetric ratcheting*. In the symmetric ratchet, participants use the old chain key to derive a new chain and message key.

5. Whenever the session changes direction (i.e., the current receiver wants to reply), both parties perform *public-key ratcheting*. In the public-key ratchet, participants use the old root key and newly sampled asymmetric key material to derive a new root key.

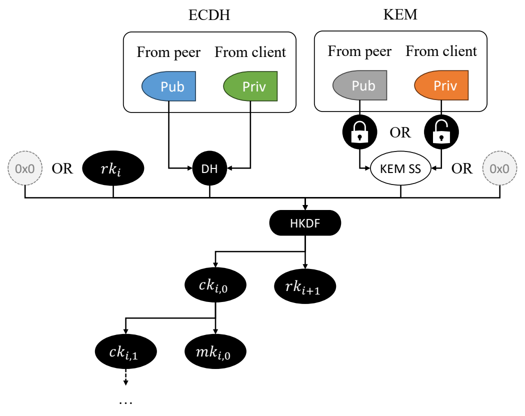


Figure 1: Dependency between the keys used by PQ3. Arrows denote that one value is used to derive another. The lock icons denote KEM encapsulation or decapsulation respectively. Sometimes a zero-byte sequence is used instead of a root key or KEM shared secret.

At this high level of abstraction, Steps 2–5 resemble the standard double-ratchet construction. But there are significant differences in the concrete details on how the ratchets are performed, in particular how a post-quantum KEM is integrated into the ratcheting.

4.2 More Detailed Account

We now expand on the above account. Although this account is more detailed, we still focus on the essential ideas and we omit some low-level details, like message and key derivation tags. Moreover, we describe some additional features of PQ3 at the end of this section. We provide a detailed specification, which includes auxiliary tags and data fields, in Appendix C.

Keys PQ3 specifies many keys. Every participant has a *long-term identity key*, a P-256 ECDSA public/private key pair to authenticate messages and other key material. Long-term identity public keys are distributed and authenticated using the IDS. All other keys are used to derive message keys. Figure 1 depicts the dependencies between these keys.

We start by introducing PQ3's three types of symmetric keys. These symmetric keys are always derived with respect to a given public-key ratchet step (identified by i in Fig. 1). *Message keys* (depicted as $mk_{i,0}$) are the message encryption keys and are derived from *chain keys* (depicted as $ck_{i,0/1}$). Chain keys are derived from either previous chain keys or initially from the same entropy sources as root keys. *Root keys* (depicted as $rk_{i/i+1}$) are used in every public-key ratchet step and, in particular, maintain the entropy from previous public-key ratchets.

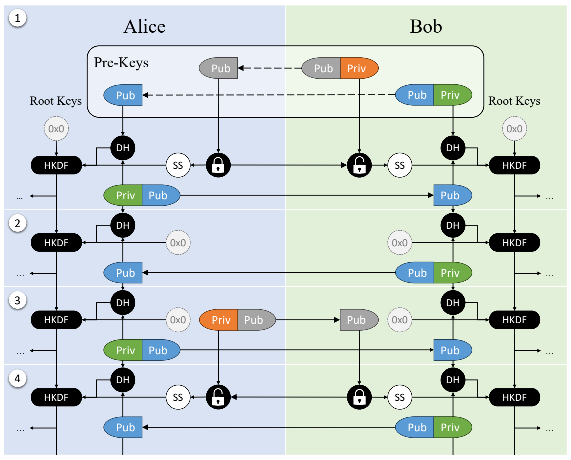


Figure 2: PQ3's public-key ratchet. Each block 1-4 illustrates a public-key ratchet step. We omit the symmetric ratchet; chain and message keys are derived from the output of the HKDF (denoted by "…"). In Step 1, Alice initiates a session with Bob and uses pre-key material (white box) to derive a root key. Alice sends a freshly encapsulated shared KEM secret (lock icon), and a freshly sampled ECDH public key to Bob that Bob can use to derive session keys. New KEM shared secrets are only encapsulated and shared when a new KEM public key was sent in the previous public-key ratchet (see block 4). Orange/gray key pairs denote ML-KEM keys, green/blue key pairs denote ECDH keys. This figure was inspired by [3].

Root and initial chain keys are derived from three entropy sources: the session's previous root key (or a zero-byte sequence upon session start; rk_i in Fig. 1), an ECDH shared secret ("DH" in Fig. 1), and optionally a KEM shared secret (replaced with a zero-byte sequence when omitted; "KEM SS" in Fig. 1). To establish these shared secrets, every client uses P-256 ECDH public/private key pairs, which we call *ECDH keys*, and ML-KEM 768 or 1024 public/private key pairs, which we call *KEM keys*. Clients establish the ECDH shared secret by combining an ECDH public key from their peer with their own ECDH private key ("ECDH Pub/Priv" in Fig. 1). Clients establish the KEM shared secret either by encapsulating it for their peer using their peer's KEM public key or by having their peer encapsulate it for them and decapsulating it with their own KEM private key ("KEM Pub/Priv" in Fig. 1).

In general, every client uses distinct, fresh ECDH and KEM keys for every session, the public part of which they send in PQ3 messages to their peer. These session-specific keys are called *ephemeral keys*. Ephemeral keys are short-lived and used only for a specific session. To support asynchronous messaging, clients use ECDH and KEM public *pre-keys* instead of their ephemeral counterparts upon session start (the ECDH and KEM keys depicted in Figure 1 could be either ephemeral or pre-keys). Clients upload their pre-keys to the IDS using timestamped pre-key bundles, which are signed with their long-term identity key. Clients can fetch their peers' pre-keys from the IDS to start a new session with any of their peers' clients without requiring that client to be online. Pre-keys can be reused in multiple sessions, but are only used upon session start. PQ3 uses ML-KEM 768 key pairs for ephemeral KEM keys and ML-KEM 1024 key pairs for KEM pre-keys.

Session Establishment In the following, we assume, as before, that Alice wishes to establish a new session with Bob. We depict an example run of PQ3 in Figure 2, specifically showing the key derivations of both parties. The figure shows four public-key ratchet steps (numbered 1-4). Step 1 illustrates session establishment as explained next. Note that all messages sent between parties include a signature by the respective sender for authentication purposes using their long-term identity key. We omit signatures, long-term identity keys, the steps of the symmetric ratchet, and sent messages from the figure to avoid clutter and to focus on the key material used in root key derivation.

Alice's actions are depicted in the left, blue half of Figure 2. Alice initiates her session with Bob by performing an IDS query for Bob's identity. Alice thereby learns three keys from the query's result: Bob's long-term identity public key, an ECDH public pre-key, and a KEM public pre-key. Querying and using pre-keys is depicted within the white box in Figure 2. Alice then generates a fresh ECDH ephemeral public/private key pair ("Priv/Pub" in Step 1) and encapsulates a fresh KEM shared secret with Bob's public pre-key (lock icon in Step 1). The encapsulation algorithm provides Alice with the cleartext KEM shared secret for her use (shown as "SS" in Step 1), and ciphertext to be given to Bob (the lock to the right of "SS", showing that it used the KEM public pre-key from above). Bob can decapsulate the KEM shared secret with his KEM private pre-key to receive the same KEM shared secret. Alice then combines her ECDH ephemeral private key with Bob's ECDH public pre-key to obtain the initial ECDH shared secret (depicted as "DH").

Alice now proceeds to derive the initial root key (and the associated initial chain key) from a zero-byte sequence, which stands in for the previous root key, together with the ECDH shared secret and the KEM shared secret, visible on the far left of Figure 2 as "HKDF" in Step 1. She uses the initial chain key to derive a message key, which she uses to encrypt her initial message. She then sends Bob the following: The ciphertext, her ECDH ephemeral public key, the KEM encapsulation (with the latter two shown in Figure 2), a hash of Bob's public pre-keys used (called the *pre-key hash*), and her signature on all these elements.

Bob uses that message to derive the initial root and chain key. Bob's actions are depicted in the right, green half of Figure 2. Bob first performs an IDS query to receive Alice's long-term identity public key (not depicted in Figure 2), which he uses to verify the message signature. Bob then looks up the private parts of his pre-keys used by Alice, which are identified by the pre-key hash. Bob decapsulates the KEM encapsulation to obtain the KEM shared secret (the open lock symbol in Step 1), and combines Alice's ECDH public ephemeral key with his ECDH private pre-key to establish the ECDH shared secret ("DH" in Step 1). With these two values (and the zero-byte sequence), Bob computes the initial root and chain key (illustrated by "HKDF" in Step 1) and derives a message key from that chain key to decrypt the ciphertext.

Symmetric Ratchet With a shared root key established, Alice can send any number of additional messages to Bob without the participants updating the root key. Nevertheless, each of these messages will be encrypted with a distinct key derived by symmetric ratcheting. Whenever a participant encrypts a message, they use the current chain key to derive a message key, and then ratchet the chain key forward by deriving a new chain key from the previous one. PQ3 establishes per-message forward secrecy as soon as the previous chain and message keys are deleted, i.e., participants should only store the latest root and chain key. The symmetric ratchet, though, is only executed as long as the conversation's direction does not change, i.e., as long as the current sender keeps sending. Whenever the current receiver wishes to respond, they perform a public-key ratchet instead.

Public-Key Ratchet Suppose, after receiving some messages from Alice, that Bob wants to reply. This means that the conversation *changes direction*, and whenever this happens clients perform the public-key ratchet. Every public-key ratchet updates the root key and derives a new, initial chain key. The steps taken to derive these new keys are similar to the steps taken during session establishment. Figure 2 illustrates (next to session establishment) three further public-key ratchet steps (numbered 2-4).

To perform the public-key ratchet, Bob first generates a fresh ECDH ephemeral public/private key pair. Depending on the conversation's state, Bob may additionally perform either of the following two actions: (i) use the encapsulation algorithm to produce a new KEM shared secret and ciphertext (for decapsulation by Alice), or (ii) generate a new KEM ephemeral public/private key pair. Action (i) is performed whenever Bob's peer, Alice, performed Action (ii) in the previous public-key ratchet. To save bandwidth, Action (ii) need not always be performed. Instead, a custom heuristic determines when a client refreshes its KEM keys. The heuristic accounts for the threat environment, performance, and other requirements. As per iOS 17.4, PQ3 clients send a fresh KEM public key roughly every 50 messages or whenever they have not sent a fresh KEM public key within a week [30].

Bob then derives the next root key and the associated initial chain key. He first combines his freshly generated ECDH ephemeral private key with Alice's ECDH ephemeral public key to obtain the new ECDH shared secret. He then uses the previous root key, the new ECDH shared secret, and either the new KEM shared secret or a zero-byte sequence (depending on whether Bob performed Action (i)) to derive the next root key and associated initial chain key. He again derives a message key from that chain key to encrypt his message and sends Alice the following values: the ciphertext, his fresh ECDH ephemeral public key, optionally the new KEM encapsulation (Action (i)), optionally his new KEM public key (Action (ii)), and a signature on all the above.

Figure 2 depicts in Step 3 that Alice generates a new ephemeral KEM public/private key pair and sends the corresponding public key to Bob, i.e., Alice executes Action (ii) above. This means that Bob will execute Action (i) in Step 4.

Overall, the cryptographic constructions used are hybrid: all key derivations incorporating a KEM shared secret also involve classical secrets. This design entails (and we establish this formally in our proofs) that PQ3's security is at least as strong as when using classical cryptography alone. The repeated use of the KEM encapsulation in the protocol therefore strictly strengthens the protocol to provide post-compromise security even against a "harvest now, decrypt later" adversary who managed to access some KEM shared secret.

5 Security Proofs

In this section, we describe how we modeled PQ3 and proved it secure using TAMARIN. In the following, we briefly introduce TAMARIN (Section 5.1), describe our protocol model (Section 5.2), the formal security properties (Section 5.3), and our proofs (Section 5.4). Our protocol model covers PQ3 in all of its complexity, i.e., its nested loops, combination of cryptographic primitives, and the different options modeled for the adversary to abuse those primitives. We discuss limitations and proof effort in Section 5.5. All our formal models and proofs are openly accessible on GitHub [10].

5.1 Background on Tamarin

TAMARIN works in the *symbolic model* of cryptography, which supports a high degree of automation when constructing proofs. TAMARIN uses labeled *multiset rewriting rules* to model setup assumptions and the behavior of protocol participants. The participants play in so-called roles, where the possible actions of each role are given by sets of rules. TAMARIN verifies security properties with respect to an active network adversary who can read, intercept, reorder, replay, and send messages. In addition to this built-in adversary, modelers can give the adversary additional capabilities using explicit rules.

Each rule has a premise and conclusion. These consist of (potentially *persistent*) *facts*, which store the terms that

TAMARIN manipulates and reasons about. The rules together specify an infinite-state transition system. Each state of this transition system includes the protocol-state associated with each role instance, the adversary's knowledge, all messages being sent on the network, and more. To apply a rule, the facts in its premise must be found in the current global state. When a rule is applied, all non-persistent facts appearing in the premise of the rule are removed from the state and instances of all facts in the conclusion of the rule are added.

All rules are labeled and TAMARIN reasons about traces, which are sequences of the instantiated rules' labels. For this, TAMARIN supports a subset of first-order logic to specify the properties one then proves. Furthermore, formulas in this logic can also be used to specify *restrictions* on which traces TAMARIN should consider when proving theorems. Restrictions can be used, for example, to state that a participant performs a certain check, e.g., signature verification, in which case traces with failed checks would be excluded.

To model different cryptographic primitives, TAMARIN supports a number of built-in equational theories, for example, for symmetric encryption and message signing. The user can additionally define their own equational theories.

TAMARIN reasons using backwards search. Starting from the protocol's specification, it negates the property to be verified and searches for a trace representing an attack. If there cannot exist any such trace, this proves the property. Internally, TAMARIN uses constraint solving, and supports both an automatic mode and an interactive mode. Each step is machine-checked, using sound and complete proof rules. However, as the underlying problem is undecidable, there is no guarantee of termination. Users can help TAMARIN construct proofs in an interactive mode, where again the prover (soundly) checks each proof step. Users can also help TAMARIN by specifying auxiliary properties that can be proven once and for all and that can be reused in larger proofs.

Finally, TAMARIN also supports a form of induction. This is essentially an induction on the length of a trace with a distinguished special *last* timepoint. Timepoints in general provide an order on the steps in the protocol. For the special last timepoint, the property must be proven, with it being assumed at all previous timepoints. We explain TAMARIN's induction scheme more detailed in Appendix A.1.1.

5.2 Protocol Model

We used TAMARIN to comprehensively model PQ3 as described in Section 4.2. Our model comprises a set of rules and restrictions, modelling PQ3 as a state transition system, and an equational theory, modelling cryptographic primitives. In this section, we describe the rules and restrictions and refer to Appendix B for details on our equational theory. The full protocol model is provided on GitHub [10].

We provide an overview of our model's protocol rules in Figure 3. Our formal model has three parts. The first part models the generation of long-term signing keys and prekeys (rule `UserKeyGen`), and IDS queries (rule `QueryIDS`). These are setup rules, which are the same for all participants, independent of whether they start a session as the sender or receiver. The second and third part model the adversary's capabilities and PQ3's protocol flow respectively.

In our model of the adversary's capabilities, we allow the adversary to compromise every private, root, chain, and message key through dedicated reveal-rules, unless our security lemmas explicitly forbid a certain key to be revealed. Additionally, we model the "harvest now, decrypt later" capability as follows. Whenever participants generate a non-post-quantum-secure key, like a fresh ephemeral ECDH private key, our model saves the key in a persistent state fact (i.e., a fact that is not consumed when it is used in a rule's premise). The adversary can then access any secrets stored this way after the rule `PQAttackerStart` is applied, but from that point on, no honest participant runs PQ3.

Our model of PQ3's protocol flow is depicted as the big blue box in Figure 3. The left-hand side depicts all sender-related rules, the right-hand side all receiver-related rules, and in the center is a `Session` fact that stores all information needed to send and receive messages. For example, a `Session` fact stores a participant's most recently generated ECDH and KEM private keys and the corresponding public keys of their peer, as well as any derived root and chain keys.

A new session is started by applying one of the rules `SessionStartAsSender` or `ReceiverStart`. These are the only two rules that only produce and do not consume a `Session` fact. Most other rules update a session, i.e., they consume and produce a `Session` fact, and they can be applied arbitrarily many times per session. After a new session has started, one of two things can happen. Either the conversation does not change direction and then both participants will apply the symmetric ratchet rules, or the conversation changes direction and the public-key ratchet rules are applied.

When being the receiver, a participant may non-deterministically choose to become sender. When they do, they perform the public-key ratchet. Depending on whether their peer had sent them a new KEM public key previously, they may additionally encapsulate a new KEM shared secret. Also, the new sender may non-deterministically send a new KEM public key themself to their peer.

A participant changes from the sender to the receiver role when they receive a new message while being in the sender state. When a participant becomes the receiver, they perform the public-key ratchet as well. In one of the two rules, they do so using a decapsulated KEM shared secret, and in the other rule they use a zero-byte sequence instead.

Intuitively, one can consider our model as implementing two nested loops. First, there is the outer, public-key ratchet loop where participants generate new ephemeral ECDH secret keys and derive root and chain keys. Second, there is the inner, symmetric ratchet loop where participants derive message

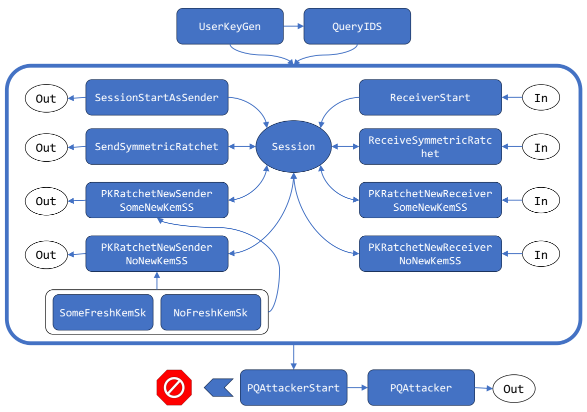


Figure 3: Overview of our formal model. Rectangles denote rules and ellipses denote facts, with their respective name printed inside. Arrows denote fact consumption and generation or rule transition. The white rectangle around `Some/NoFreshKemSk` denotes that either of the rules is applied non-deterministically. The rule `PQAttackerStart` can be applied at any point. When this happens, protocol execution halts (modeling a "harvest now, decrypt later" adversary) and thereafter the rule `PQAttacker` can be applied, which reveals any non-post-quantum-secure secret to the adversary. This figure omits rules that reveal key material.

keys and send messages. The symmetric ratchet loop always runs within one iteration of the public-key ratchet loop.

5.3 Properties Specified

5.3.1 Secrecy

PQ3 aims to satisfy three secrecy properties: message secrecy, forward secrecy, and post-compromise security. In our formalization, we combine all three into a single property. This property is formulated as a formula, called a *lemma* in TAMARIN, as one must prove that it holds for the protocol.

Figure 4 contains our secrecy lemma.[1] It states that the adversary cannot know a message (line 4) that has been previously sent (line 2), unless the adversary succeeds in at least one of four kinds of compromise, listed below. The kinds of compromise are formulated with respect to the keys referenced by the `SessionSecrets` fact. This fact lists all keys and shared secrets used by the sender when sending the respective message, e.g., their most recently sampled ephemeral ECDH public key (`myEcdhPk`) and the most recently encapsulated KEM shared secret (`kemSS`). We sketch a possible attack for each kind of compromise to show that our secrecy lemma is *tight*. Namely, dropping any but the first disjunct yields a counterexample. To learn a message sent with PQ3, the adversary must compromise at least:

- The message key used during encryption from either the receiver or sender (line 5 in Figure 4). Should the adversary learn the message key, they could simply decrypt the message themself.

- One of the chain keys used in the symmetric ratchet to derive the message key from either the receiver or sender (lines 6-7; `a << b` denotes that `a` is a *subterm* of `b` [31]). Should the adversary learn one of these chain keys, they could simply derive the message key themself.

[1] In the following, we will sometimes shorten the names of facts in lemmas when compared to the source files, e.g., `RevealIdentityKey` may become `RevealIDKey`.

- The recipient's long-term identity key before the message `msg` was sent (line 8). In this case, the adversary could generate a fresh ECDH ephemeral and KEM encapsulation key and send them to the messaging partner in question. This attack allows the adversary to carry out all communication in their victim's stead.

- One of the ephemeral ECDH secret keys, used to derive the most recently established ECDH shared secret, *and* the KEM shared secret (lines 9-14). This allows the adversary to perform a public-key ratchet step themself.

  The adversary can learn an ECDH secret key either through direct compromise (lines 10-11) or using a quantum computing attack when a sufficiently powerful quantum computer becomes available (line 9). The compromise of the sender's ECDH pre-key has no effect because a sender will always sample a fresh ECDH ephemeral key upon session start.

  The KEM shared secret can be effectively compromised in two ways. First, the adversary can compromise the KEM secret key used for encapsulation (lines 12-13). Second, the adversary can circumvent the need to directly compromise the KEM shared secret by compromising a root key derived after that KEM shared secret was established (line 14). In the latter case, if the adversary additionally learns an ECDH secret key used in a subsequent public-key ratchet step, they can derive the respective initial chain key themself.

  In addition to the ECDH and KEM shared secret, the adversary also requires the root key from the previous public-key ratchet to perform the current public-key ratchet themself. Our threat model, however, permits this root key to be revealed to the adversary in general.

Our protocol model allows the adversary to access all private, message, chain, and root keys unless a lemma explicitly forbids it. In particular, our secrecy lemma only forbids the adversary to access key material related to the key material used for sending the message. All key-reveal assumptions in lines 5-14 refer to the key material referenced in line 3, which

```
1  All id me them msg myEcdhPk theirEcdhPk kemSS encapPk chainKey msgKey #t.
2        ( Sent(id,_,me,them,msg) @ t
3        & SessionSecrets(myEcdhPk,theirEcdhPk,kemSS,encapPk,chainKey,msgKey) @ t)
4    ==>   (not Ex #x. K(msg) @ x)
5        | (Ex #x. RevealMessageKey(me,msgKey) @ x) | (Ex #x. RevealMessageKey(them,msgKey) @ x)
6        | (Ex ckC #x. RevealChainKey(me,ckC) @ x & (ckC << chainKey | ckC = chainKey))
7        | (Ex ckC #x. RevealChainKey(them,ckC) @ x & (ckC << chainKey | ckC = chainKey))
8        | (Ex #x. RevealIDKey(them) @ x & #x < #t)
9        | ( ( (Ex #x. PQAttack() @ x)
10           | (Ex #x. RevealECDHPreKey(them,theirEcdhPk) @ x)
11           | (Ex #x. RevealECDHKey(id,me,myEcdhPk) @ x) | (Ex #x. RevealECDHKey(_,them,theirEcdhPk) @ x))
12         & ( (Ex #x. RevealKemKey(me,encapPk) @ x) | (Ex #x. RevealKemKey(them,encapPk) @ x)
13           | (Ex #x. RevealKemPreKey(me,encapPk) @ x) | (Ex #x. RevealKemPreKey(them,encapPk) @ x)
14           | (Ex k #x. RevealRootKey(me,kemSS,k) @ x) | (Ex k #x. RevealRootKey(them,kemSS,k) @ x)))
```

Figure 4: Secrecy lemma. The lemma formalizes that if a message `msg` was sent using the secret values referenced by `SessionSecrets`, then either the message cannot be known by the adversary (line 4) or the adversary compromised a specific combination of keys (lines 5ff.). Section 5.3.1 explains this lemma, line-by-line, in further details.

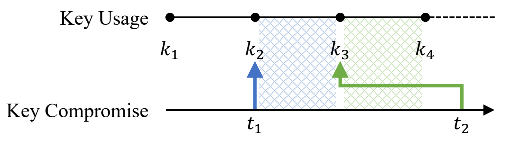


Figure 5: A participant derives four initial chain keys (k_1-k_4) over time and the adversary compromises k_2 and k_3 at times t_1 and t_2 respectively. No matter at which point in time (compare t_1 with t_2), key compromise has similar and limited effect: The adversary can only learn messages sent before the next initial chain key is derived (the shaded areas).

in turn is bound to the `Sent` event in line 2 (by the temporal variable `t`). Thus, proving secrecy establishes forward secrecy and post-compromise security as we explain next.

For long-term identity keys, we show that PQ3 provides forward secrecy in that all messages exchanged prior to the compromise of such a key remain secure (see line 8). For most encryption keys (exceptions and details below), we establish forward secrecy and post-compromise security in that to compromise some given message, the adversary must learn the respective key used for that message and the compromise of past or future keys has no effect. Note that “past” and “future” here refer to the points in time when a key was used, not when it was compromised. In particular, this allows us to establish post-compromise security guarantees even *after* the adversary obtained a quantum computer and participants stopped running the protocol. Although participants will no longer rotate keys, as they no longer run the protocol, they will have self-healed from the compromise of any other key than the one used most recently. For an illustration, see Figure 5.

For some keys, forward secrecy and post-compromise security are only established under further constraints. In these cases, our secrecy lemma precisely defines the point in time at which forward-secrecy or post-compromise security are established. We list all forward secrecy and post-compromise security guarantees entailed by our secrecy lemma below and, wherever necessary, describe the constraints on these guarantees. When the adversary does not possess a quantum computer, PQ3 provides:

- Forward secrecy w.r.t. either peer’s long-term identity keys.
- Forward secrecy and post-compromise security w.r.t. ECDH ephemeral keys.
- Post-compromise security w.r.t. ECDH pre-keys as soon as a new ECDH ephemeral key is generated by a session’s initial recipient.

In practice, PQ3 also provides forward secrecy for ECDH pre-keys as it requires that participants update their pre-keys registered at the IDS every 2 weeks. As soon as a client registers a new pre-key, they establish forward secrecy for all previous session-start messages sent to them.

Should the adversary at some point break all non-ML-KEM keys using a quantum computer, PQ3 still provides:

- Post-quantum forward secrecy and post-compromise security w.r.t. ML-KEM keys.
- Forward secrecy and post-compromise security w.r.t. chain and message keys. For chain keys, PQ3 establishes post-compromise security upon the next public-key ratchet against chain key compromise. PQ3 establishes both properties even when the adversary possesses a quantum-computer because these keys depend on KEM-encapsulated secrets.

Note that working out and rigorously proving such fine-grained notions of secrecy is nontrivial and one strongly benefits here from a proof assistant. Overall, our TAMARIN proof of secrecy establishes that, in the absence of the sender or recipient being compromised, all keys and messages transmitted are secret. The secrecy property is fine-grained in that compromises can be tolerated in a well-defined sense where

```
All id i s r m #t. Received(id, i, s, r, m) @ #t
==> ( (Ex #x. Sent(_, i, s, r, m) @ #x & #x<#t)
    | (Ex #x. RevealIDKey(s) @ #x & #x<#t))
```

Figure 6: Agreement lemma. This formalizes that for every message-receive event, there must be a corresponding message-send event for which the participants agree on the sender, receiver, and message counter, unless the sender's long-term identity key was previously compromised.

```
1  All s1 s2 r1 r2 m rEcdhPk1 mk1 rEcdhPk2 mk2 #t1 #t2.
2      ( Received(_, _, s1, r1, m) @ #t1
3      & SessionSecrets(rEcdhPk1, _, _, _, mk1) @ #t1
4      & Received(_, _, s2, r2, m) @ #t2
5      & SessionSecrets(rEcdhPk2, _, _, _, mk2) @ #t2)
6  ==> ( (#t1 = #t2)
7      | ( rEcdhPk1 = rEcdhPk2 & mk1 = mk2
8        & s1 = s2 & r1 = r2
9        & Ex #x. ECDHPreKeyGen(r1, rEcdhPk1) @ #x)
10     | (Ex #x. RevealIDKey(s1) @ #x & #x < #t1 )
11     | (Ex #x. RevealIDKey(s2) @ #x & #x < #t2))
```

Figure 7: Injective agreement lemma. It formalizes that for two message-receive events of the same message `m`, either these events must be the same (line 6), or they were sent using the recipients pre-key (lines 7f.), or one sender's identity key was compromised (lines 10ff.).

the effect of the compromise on the secrecy of data is limited in time and effect as detailed above. Moreover, we show that PQ3 combines the security of both classical and post-quantum-secure cryptographic primitives. Hence, to break PQ3 one must break both.

5.3.2 Agreement

In contrast to secrecy, formalizing agreement is much simpler. This is because PQ3 relies on the participants' long-term identity keys' security to provide agreement. Compromise of a participant's long-term identity key is both necessary and sufficient to break agreement. It is necessary because an attacker must generate a message signature when trying to spoof a sender, and it is sufficient because a sender need not compromise the sender's encryption keys to send an inauthentic message; they can simply generate their own and send them alongside the faked message.

Our formalization of agreement is split into two TAMARIN lemmas (Figures 6 and 7). The first lemma formalizes agreement: Whenever a participant `r` receives a message `m`, apparently from `s` and with message counter `i`, then either `s` had previously sent `m` to `r` with counter `i` or that senders' long-term identity has been compromised in the past.

The second lemma formalizes that the agreement is injective [25], meaning that there is a one-to-one mapping from receive-events to send-events. This lemma states that for every two honest message-receive events of the same message, these events must either be the same (`#t1 = #t2`), or a recipient's ECDH pre-key rather than an ephemeral key was used to derive the message key (lines 7-9), or either of the senders were compromised. Compromise of one sender suffices to violate injective agreement because agreement does not entail secrecy. The adversary could learn a message by compromising the ECDH and KEM keys of the session. They could then send the message again, which requires the compromise of a long-term identity key, however, to produce the necessary signature.

During our proof efforts, we noticed a trivial violation of injective agreement, which is covered by lines 7-9. Clearly, PQ3 cannot provide injective agreement for session-start messages (and messages sent as part of the symmetric ratchet directly thereafter) as pre-keys can be reused for session starts. Thus, recipients will accept session-start messages multiple times. In practice, this case must be addressed by the session-handling layer, which defines under which conditions clients will accept session-start messages from devices they already have an existing session with. We shared this finding with Apple researchers who confirmed that their session-handling layer indeed addresses this case. Put differently, our formal proofs highlight precisely the assumptions on the session-handling layer needed to securely deploy PQ3.

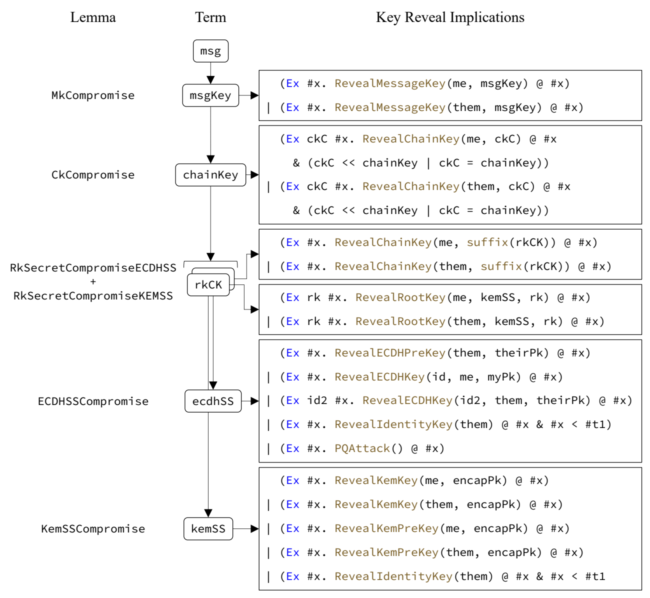
5.4 Proofs & Proof Methodology

We describe here our proofs and proof methodology for PQ3. Our proof methodology applies to theories that include (possibly nested) loops and for which trace formulas like secrecy or authentication are to be proven. We present our methodology more generally and with further details in Appendix A.

We encountered two challenges when verifying PQ3. First, PQ3 employs a nested loop. If not carefully handled, loops result in prover non-termination as they are unrolled infinitely often. TAMARIN provides induction to address this problem, but using induction correctly, especially when loops are nested, requires postulating nontrivial auxiliary lemmas.

Second, our threat model considers the leakage of "synthetic" key material, derived using a KDF, and our lemmas naturally must refer to this key material. When proving secrecy, we repeatedly encountered cases similar to the following. TAMARIN would consider an honest session sending a message, claiming that the adversary could get the decryption key for this message (violating secrecy) from a completely unrelated session. We call such unrelated sessions *ghost sessions*. In this case, the non-trivial proof goal was to convince TAMARIN that the ghost session must be the same as the honest session or the peer's session. Note that other protocol models typically only consider the leakage of "atomic" key material, i.e., key material modelled as a fresh term.

To address these two challenges, our methodology uses three kinds of auxiliary lemmas.



```
  (Ex #x. RevealMessageKey(me, msgKey) @ #x)
| (Ex #x. RevealMessageKey(them, msgKey) @ #x)
```

```
  (Ex ckC #x. RevealChainKey(me, ckC) @ #x
    & (ckC << chainKey | ckC = chainKey))
| (Ex ckC #x. RevealChainKey(them, ckC) @ #x
    & (ckC << chainKey | ckC = chainKey))
```

```
  (Ex #x. RevealChainKey(me, suffix(rkCK)) @ #x)
| (Ex #x. RevealChainKey(them, suffix(rkCK)) @ #x)
```

```
  (Ex rk #x. RevealRootKey(me, kemSS, rk) @ #x)
| (Ex rk #x. RevealRootKey(them, kemSS, rk) @ #x)
```

```
  (Ex #x. RevealECDHPreKey(them, theirPk) @ #x)
| (Ex #x. RevealECDHKey(id, me, myPk) @ #x)
| (Ex id2 #x. RevealECDHKey(id2, them, theirPk) @ #x)
| (Ex #x. RevealIdentityKey(them) @ #x & #x < #t1)
| (Ex #x. PQAttack() @ #x)
```

```
  (Ex #x. RevealKemKey(me, encapPk) @ #x)
| (Ex #x. RevealKemKey(them, encapPk) @ #x)
| (Ex #x. RevealKemPreKey(me, encapPk) @ #x)
| (Ex #x. RevealKemPreKey(them, encapPk) @ #x)
| (Ex #x. RevealIdentityKey(them) @ #x & #x < #t1
```

Figure 8: Connection of Adversary-Construction Lemmas for Message Secrecy. Arrows denote logical implication.

Loop-Jump Lemmas These lemmas allow one to skip unrolling the steps of a (nested) loop and jump to a "relevant" point in a loop, for example, its beginning or where a specific term was introduced.

Variable-Linking Lemmas These lemmas establish that for two instances of the same fact using two variables *a* and *b*, if both facts have the same value for *a*, they must have the same value for *b*.

Adversary-Construction Lemmas These lemmas formalize how an adversary could construct a term. Typically, the adversary can either construct it or access it using a dedicated reveal rule (which in turn typically implies a contradiction to the threat model). Figure 8 depicts our model's adversary-construction lemmas. For example, `CkCompromise` states that the adversary can only know a chain key if they know the value that gets split into the root and chain key (`rkCK`), or they compromised this or a previous chain key.

Loop-jump lemmas are the foundation for proving properties of models including nested loops. Without such lemmas, TAMARIN's induction fails to prove even the simplest properties of an outer loop. The induction hypothesis will not apply in cases where a step in the outer loop is directly preceded by a step in an inner loop. Moreover, adversary-construction lemmas are required to deal with the complicated terms that are computed in nested loops, and variable-linking lemmas are required to address ghost sessions.

We proved secrecy for PQ3 using a series of adversary-construction lemmas, depicted in Figure 8, which in turn were proven using the loop-jump and variable-linking lemmas in Figures 9 and 10. Concretely, when proving secrecy,

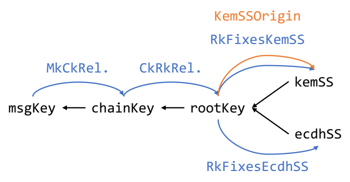


Figure 9: Loop-Jump (orange) and Variable-Linking Lemmas (blue) Related to Key Derivation. Black arrows indicate which variables are used to construct other variables, e.g., a message key is derived from a chain key.

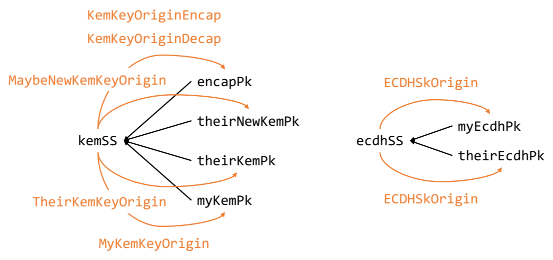


Figure 10: Loop-Jump Lemmas (orange) Related to Establishing Shared Secrets. Black arrows indicate which key material can be used to establish which shared secret.

TAMARIN first negates the original lemma and tries to construct a trace satisfying the negated lemma, i.e., TAMARIN tries to construct a trace where a message has been sent and the adversary knows it. By solving for how the adversary could learn the message, TAMARIN deduces that the adversary must know the message key used for encryption. This allows us to apply the first adversary-construction lemma `MkCompromise`. This lemma expresses that the adversary can only know the message key if they either know the respective chain key (allowing us to apply the next adversary-construction lemma) or if they access a reveal rule (contradicting our threat model assumptions directly). In the case where the adversary knows a respective smaller term, we can apply the next adversary-construction lemma, etc. Finally, the lemmas `ECDHSSCompromise` and `KemSSCompromise` directly contradict the threat model.

We proved these adversary-construction lemma using the loop-jump and variable-linking lemmas depicted in Figures 9 and 10. A sequence of variable-linking lemmas (depicted in blue) connect message to chain to root keys and to the respective KEM shared secret and ECDH shared secret (Figure 9). Loop-jump lemmas (depicted in orange) then connect the shared secrets to the asymmetric key material used to establish them (Figure 10). This allows TAMARIN to deduce that access to the shared secret requires access to the respective private key material. Beyond the lemmas depicted in Figures 9 and 10, we only use the three loop-jump lemmas `RootKeyConnectionReceive`, `RootKeyConnectionSend`, and `SessionStart`, which jump from an instance of the symmetric ratchet to the most recent public key ratchet (switching from sender to receiver or receiver to sender respectively) and the session start.

We proved both agreement lemmas much like we proved secrecy, but proving agreement was much simpler. PQ3 provides agreement by signing every message. When trying to prove non-injective agreement, TAMARIN immediately finds that to violate agreement, the adversary must generate this signature themself, which in turn requires access to the signing key. The rule that introduces the signing key, however, can directly be established using the `SessionStart` lemma as signing keys are queried only upon session start.

When attempting to prove injective agreement, TAMARIN will start by constructing a trace with two honest receive events for the same message. Using variable-linking lemmas, we can establish that these two sessions must use the same ECDH shared secret, and using the respective loop-jump lemmas, we can jump to the rule instantiation where the receiver generated their latest ECDH ephemeral key. This allows TAMARIN to derive that the two receive events must have happened in the same session (unless a pre-key was used; but this case is addressed in the lemma directly).

Finally, we only use six auxiliary lemmas not fitting the categories defined above. These lemmas simply limit TAMARIN's search space to reduce proof construction time. For example, they show that certain events (like session start) can only occur once, or establish well-formedness conditions (for example, that the root key is a subterm of the chain key).

5.5 Discussion

5.5.1 Scope of Analysis

Our analysis covers the protocol design as described in the documentations we received from Apple. PQ3's implementation is not part of our analysis. Furthermore, as our analysis is based on symbolic models, it abstracts away some details of the concrete implementation, like message lengths and some algorithmic details of the ciphers used.

Our model does not consider session management, it does not consider long-term identity or pre-key rollover, and it only considers group messaging implicitly. However, PQ3 implements group messaging using multiple, individual device-to-device sessions, and our analysis establishes the security of each such session. We did not model session management as a specification of iMessage's session management was not available to us. Moreover, PQ3 is not limited in its use to iMessage. Different applications may have different requirements on their session handling. Studying PQ3 in isolation is therefore desirable in its own right. Finally, PQ3 specifies an authenticated data field for messages, however, puts exact values as out-of-scope. We did not model this field explicitly as our model proves agreement on the entire message (thus subsuming authenticated data).

Beyond the limitations just mentioned, our formal model incorporates all details that were part of the documentation provided to us by Apple. In particular, we did not abstract away any protocol steps that participants would take.

5.5.2 Proof Effort

Our TAMARIN model comprises 32 lemmas in total. Next to the auxiliary lemmas used to prove secrecy and agreement (Section 5.4), our model includes a *sources lemma*, which aids TAMARIN in precomputation steps, and two *executability lemmas*. Executability lemmas effectively "sanity check" a protocol specification by establishing that the participants can run the protocol without adversary involvement. This enhances our confidence that the protocol model faithfully represents the protocol and that its properties do not hold trivially.

All proofs are guided by custom proof heuristics and finding the right heuristics to successfully construct proofs required substantial efforts for many lemmas. For example, checking the proof for the lemma formalizing injective agreement (Section 5.3.2) takes around 7 hours and requires 20 GB of RAM on a server using an Intel Xeon CPU E5-2650 v4 @ 2.20GHz. The proofs of other lemmas require up to 100 GB of RAM to be checked. Overall, we estimate that proving PQ3 took around 2.5 person-months of work.

6 Conclusions

We have used TAMARIN to formally verify the device-to-device messaging protocol PQ3. Our analysis is based on machine-checked proofs of fine-grained secrecy and authentication properties. This provides a high degree of assurance that PQ3 functions securely against an active network adversary who can selectively compromise parties, even when sufficiently powerful quantum computers become available. Additionally, the properties we prove give a detailed account of the impact that the compromise of every individual key has. Lastly, we show that TAMARIN is up to the task of reasoning about complex protocols with nested loops, and we have given a general methodology for doing this.

Future work Of particular interest would be the formal analysis of PQ3's session handling layer. [27] established that Signal may not provide post-compromise security in practice due to its implementation of session handling (see Section 2.2). Whether the same applies to PQ3 remains an open question. Furthermore, our formal model could be extended to account for IDS key roll-over, i.e., of long-term identity and pre-keys, and it could be extended to incorporate enhanced models of cryptographic primitives, such as those suggested by [32, 33, 34, 35].

References

[1] N. Borisov, I. Goldberg, and E. Brewer, "Off-the-record communication, or, why not to use pgp," in *Proceedings of the 2004 ACM Workshop on Privacy in the Electronic Society*, ser. WPES '04. New York, NY, USA: Association for Computing Machinery, 2004, p. 77–84. [Online]. Available: https://doi.org/10.1145/1029179.1029200

[2] V. Moscaritolo, G. Belvin, and P. Zimmermann, "Silent Circle Instant Messaging Protocol," Tech. Rep., Dec. 2012. [Online]. Available: https://netzpolitik.org/wp-upload/SCIMP-paper.pdf

[3] T. Perrin and M. Marlinspike, "The double ratchet algorithm," 2016, revision 1, 2016-11-20. [Online]. Available: https://signal.org/docs/specifications/doubleratchet/doubleratchet.pdf

[4] E. Kret and R. Schmidt, "The pqxdh key agreement protocol," 2023, revision 2. [Online]. Available: https://signal.org/docs/specifications/pqxdh/pqxdh.pdf

[5] T. P. Moxie Marlinspike, "The X3DH key agreement protocol," revision 1, 2016-11-04. [Online]. Available: https://signal.org/docs/specifications/x3dh

[6] M. Mosca and M. Piani, "Quantum threat timeline report," Global Risk Institute, December 2023. [Online]. Available: https://globalriskinstitute.org/publication/2023-quantum-threat-timeline-report

[7] National Institute of Standards and Technology, "Module-lattice-based key-encapsulation mechanism standard," August 2023. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.203.ipd.pdf

[8] B. Schmidt, S. Meier, C. Cremers, and D. A. Basin, "Automated analysis of Diffie-Hellman protocols and advanced security properties," in *25th IEEE Computer Security Foundations Symposium, CSF 2012, Cambridge, MA, USA, June 25-27, 2012*. IEEE Computer Society, 2012, pp. 78–94. [Online]. Available: https://doi.org/10.1109/CSF.2012.25

[9] S. Meier, B. Schmidt, C. Cremers, and D. A. Basin, "The TAMARIN prover for the symbolic analysis of security protocols," in *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, ser. Lecture Notes in Computer Science, vol. 8044. Springer, 2013, pp. 696–701. [Online]. Available: https://doi.org/10.1007/978-3-642-39799-8_48

[10] "Tamarin-pq3," tamarin-pq3, May 2024. [Online]. Available: https://github.com/tamarin-pq3/tamarin-pq3

[11] K. Bhargavan, A. Bichhawat, Q. Do, P. Hosseyni, R. Küsters, G. Schmitz, and T. Würtele, "DY*: A Modular Symbolic Verification Framework for Executable Cryptographic Protocol Code," in *6th IEEE European Symposium on Security and Privacy (EuroS&P)*, Sep. 2021. [Online]. Available: https://hal.inria.fr/hal-03178425

[12] N. Unger, S. Dechand, J. Bonneau, S. Fahl, H. Perl, I. Goldberg, and M. Smith, "SoK: Secure messaging," in *2015 IEEE Symposium on Security and Privacy*, 2015, pp. 232–249.

[13] K. Cohn-Gordon, C. Cremers, and L. Garratt, "On post-compromise security," in *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*, 2016, pp. 164–178.

[14] O. Blazy, I. Boureanu, P. Lafourcade, C. Onete, and L. Robert, "How fast do you heal? A taxonomy for post-compromise security in secure-channel establishment," in *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 5917–5934. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/blazy

[15] J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehle, "CRYSTALS - Kyber: a CCA-secure moule-lattice-based KEM," in *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, 2018, pp. 353–367.

[16] K. Bhargavan, C. Jacomme, F. Kiefer, and R. Schmidt, "Formal verification of the PQXDH post-quantum key agreement protocol for end-to-end secure messaging," in *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*, D. Balzarotti and W. Xu, Eds. USENIX Association, 2024. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/bhargavan

[17] R. Fiedler and F. Günther, "Security analysis of Signal's PQXDH handshake," Cryptology ePrint Archive, Paper 2024/702, 2024. [Online]. Available: https://eprint.iacr.org/2024/702

[18] N. Unger and I. Goldberg, "Deniable key exchanges for secure messaging," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 1211–1223. [Online]. Available: https://doi.org/10.1145/2810103.2813616

[19] B. Blanchet, "A computationally sound mechanized prover for security protocols," Cryptology ePrint Archive, Paper 2005/401, 2005. [Online]. Available: https://eprint.iacr.org/2005/401

[20] A. Bienstock, J. Fairoze, S. Garg, P. Mukherjee, and S. Raghuraman, "A more complete analysis of the signal double ratchet algorithm," in *Advances in Cryptology – CRYPTO 2022*, Y. Dodis and T. Shrimpton, Eds. Cham: Springer Nature Switzerland, 2022, pp. 784–813.

[21] R. Canetti, P. Jain, M. Swanberg, and M. Varia, "Universally composable end-to-end secure messaging," in *Advances in Cryptology – CRYPTO 2022*, Y. Dodis and T. Shrimpton, Eds. Cham: Springer Nature Switzerland, 2022, pp. 3–33.

[22] K. Cohn-Gordon, C. Cremers, B. Dowling, L. Garratt, and D. Stebila, "A formal security analysis of the signal messaging protocol," *J. Cryptol.*, vol. 33, no. 4, p. 1914–1983, oct 2020. [Online]. Available: https://doi.org/10.1007/s00145-020-09360-1

[23] J. Alwen, S. Coretti, and Y. Dodis, "The double ratchet: Security notions, proofs, and modularization for the signal protocol," in *Advances in Cryptology – EUROCRYPT 2019*, Y. Ishai and V. Rijmen, Eds. Cham: Springer International Publishing, 2019, pp. 129–158.

[24] D. Stebila, "Security analysis of the iMessage PQ3 protocol," Cryptology ePrint Archive, Paper 2024/357, 2024. [Online]. Available: https://eprint.iacr.org/2024/357

[25] G. Lowe, "A hierarchy of authentication specification," in *10th Computer Security Foundations Workshop (CSFW '97), June 10-12, 1997, Rockport, Massachusetts, USA*, 1997, pp. 31–44. [Online]. Available: https://doi.org/10.1109/CSFW.1997.596782

[26] B. Blanchet, "Automatic verification of security protocols in the symbolic model: The verifier ProVerif," in *Foundations of Security Analysis and Design VII: FOSAD 2012/2013 Tutorial Lectures*. Cham: Springer International Publishing, 2014, pp. 54–87. [Online]. Available: https://doi.org/10.1007/978-3-319-10082-1_3

[27] C. Cremers, C. Jacomme, and A. Naska, "Formal Analysis of Session-Handling in Secure Messaging: Lifting Security from Sessions to Conversations," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 1235–1252. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/cremers-session-handling

[28] N. Kobeissi, K. Bhargavan, and B. Blanchet, "Automated verification for secure messaging protocols and their implementations: A symbolic and computational approach," in *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, 2017, pp. 435–450.

[29] Apple Security Engineering and Architecture (SEAR), "Advancing iMessage security: iMessage Contact Key Verification - Apple Security Research," February 2024. [Online]. Available: https://security.apple.com/blog/imessage-contact-key-verification/

[30] F. Jacobs, "Invited talk: Designing iMessage PQ3: Quantum-Secure Messaging at Scale," Toronto, Canada, Mar. 2024. [Online]. Available: https://www.youtube.com/watch?v=RVbHElGe518

[31] C. Cremers, C. Jacomme, and P. Lukert, "Subterm-Based Proof Techniques for Improving the Automation and Scope of Security Protocol Analysis," in *2023 IEEE 36th Computer Security Foundations Symposium (CSF)*, Jul. 2023, pp. 200–213.

[32] D. Jackson, C. Cremers, K. Cohn-Gordon, and R. Sasse, "Seems Legit: Automated Analysis of Subtle Attacks on Protocols that Use Signatures," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19. New York, NY, USA: ACM, Nov. 2019, pp. 2165–2180.

[33] C. Cremers, A. Dax, and N. Medinger, "Keeping Up with the KEMs: Stronger Security Notions for KEMs and automated analysis of KEM-based protocols," 2023. [Online]. Available: https://eprint.iacr.org/2023/1933

[34] V. Cheval, C. Cremers, A. Dax, L. Hirschi, C. Jacomme, and S. Kremer, "Hash Gone Bad: Automated discovery of protocol attacks that exploit hash function weaknesses," in *32nd USENIX Security Symposium*, Anaheim, CA, USA, 2023-08-09/2023-08-11, pp. 5899–5916. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/cheval

[35] C. Cremers, A. Dax, C. Jacomme, and M. Zhao, "Automated Analysis of Protocols that use Authenticated Encryption: How Subtle AEAD Differences can impact Protocol Security," in *32nd USENIX Security Symposium*, Anaheim, CA, USA, 2023-08-09/2023-08-11, pp. 5935–5952. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/cremers-protocols

[36] H. Krawczyk, "Cryptographic extraction and key derivation: The HKDF scheme," in *Advances in Cryptology – CRYPTO 2010*, T. Rabin, Ed. Berlin, Heidelberg: Springer, 2010, pp. 631–648.

A Proof Methodology

In Section 5.4, we presented our proof methodology, specialized to how we applied it to the PQ3 Messaging Protocol. In this section, we will describe this methodology in more detail and in its generality: i.e., how one could apply it to other protocols with similar structure.

TAMARIN provides general support for handling loops, based on induction and injective facts, and we begin our account by explaining them. We afterwards introduce two minimal TAMARIN theories that illustrate the issues of nested loops and ghost sessions (see Section 5.4), but on a smaller and simpler scale. These theories will also help to illustrate the reasoning behind loop-jump, adversary-construction and variable-linking lemmas that we have seen. Finally, we generalize the resulting proof methodology.

A.1 Handling Loops in TAMARIN

A.1.1 Induction

TAMARIN can prove any formula directly by backward search, as explained in Section 5.1, or by induction. When TAMARIN proves a formula φ by induction, it rewrites it into the form

$$\mathrm{BC}(\varphi) \wedge (\mathrm{IH}(\varphi) \implies \varphi).$$

The first conjunct, $\mathrm{BC}(\varphi)$, is the base case, and it requires proving φ on an empty trace. The second conjunct, $\mathrm{IH}(\varphi) \implies \varphi$, is the induction step, which requires proving φ on the last element of the trace with φ being assumed on all previous steps of the trace. $\mathrm{BC}(\varphi)$ is defined as φ but every formula of the form $f@i$ is replaced with $\perp$. For example, for a formulation of secrecy such as

$$\forall m,t.\mathit{Sent}(m)@t \implies \neg(\exists x.K(m)@x)$$

this replacement results in

$$\forall m,t.\perp \implies \neg(\exists x.\perp).$$

$\mathrm{IH}(\varphi)$ is defined as φ but every quantified temporal variable is asserted to not be the last time point. This is done using the special predicate last, which is true if and only if it is provided the last time point as argument. For example, the induction hypothesis of secrecy as defined above would become

$$\forall m,t.\mathit{Sent}(m)@t \implies \neg(\exists x.K(m)@x \wedge \neg\mathsf{last}(x)) \vee \mathsf{last}(t).$$

After having translated φ into its inductive form, TAMARIN will attempt to prove it as any other formula. Moreover, it will attempt to prove the base case and induction step separately, and the induction hypothesis IH will become available (like an auxiliary lemma) in the branch proving the induction step.

In practice, induction is used to prove properties of protocols with loops. However, one can only prove properties of loops by induction where the loops are expressed in terms of facts that appear repeatedly in the protocol's trace. Take the above translation of secrecy as an example. In the induction step, the induction hypothesis will become effectively vacuous as long as the fact $\mathit{Sent}(m)$ only occurs at a last time point

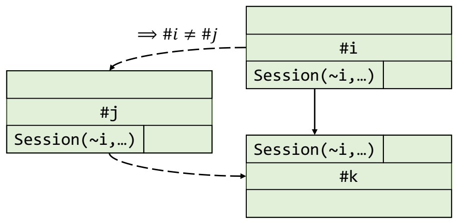


Figure 11: Illustration of a contradiction from an injective instance. The solid arrow indicates premise consumption. Dashed arrows indicate time ordering, i.e., the rule at `#j` must be applied after `#i` but before `#k`. The order of time points requires that `#j` and `#i` must be unified as `#k` consumes a session fact with the matching ID `~i`. However, `#j` must occur strictly after `#i`, which contradicts this unification occurence. There is a symmetric case where `#j` and `#k` must be unified because `#j` and `#i` share a premise.

t. In that case, the second disjunct on the right-hand side of the implication will apply, whereas one usually requires the first disjunct to apply to make progress on a proof. Only when we can introduce a new *Sent* fact in the trace that does not occur at the last time point can we use the induction hypothesis.

In particular and applied to loops, this means two things. (1) Induction can only be applied to formulas that express invariants of loops, but not to express something that holds after a loop has stopped. A loop will only end once, which makes it impossible to introduce a second end of the loop not occurring at the last time point. (2) Induction cannot be used to prove properties for outer loops without further auxiliary lemmas. When we attempt to prove properties of outer loops, TAMARIN will always also consider the case that a step in the outer loop was preceded by an inner loop of unbounded length. Also in these cases, the fact referenced in the induction hypothesis (the outer loop step) will only occur at the last time point. For both of these reasons, induction must be applied with care and cannot be blindly applied to prove arbitrary formulas of protocols with loops.

A.1.2 Injective Facts

Injective facts are commonly used to model loops in TAMARIN. They are defined as facts that (for a fixed first argument) can occur only once in the global state. We call an injective fact's first argument its *loop identifier*. If a fact satisfies the following constraints, it is automatically detected as injective by TAMARIN:

- It is not a persistent fact.
- Its loop identifier is a fresh term.
- It never occurs more than once in a rule's conclusion with the same loop identifier.

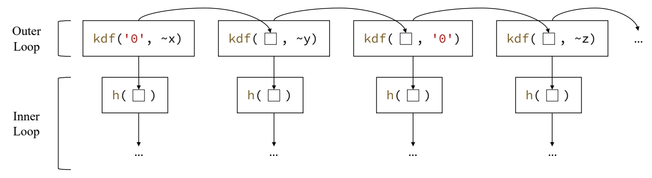


Figure 12: Nested loop example.

- Whenever it occurs in a rule's conclusion, either (a) its loop identifier is freshly generated (the loop has started), or (b) it occurs in the rule's premise with the same loop identifier (a loop step is taken).

Injective facts allow TAMARIN to derive contradictions by exploiting that all injective facts with a shared loop identifier must be linearizable. In particular, a loop step can never occur between two directly connected loop steps (as illustrated in Figure 11). It is possible to prove properties of loops without using injective facts, but using injective facts can drastically simplify proofs, so it is generally advisable to make use of this heuristic.

A.2 Proof Methodology by Example

In what follows, we illustrate the challenges encountered when constructing proofs about nested loops using three simple, minimal theories. We will introduce these theories and our proof methodology on an intuitive level. For full details, see our repository that provides all the theory files [10].

Nested Loop Example Figure 12 provides an illustration of a nested loop from the `nested-loop` theory in [10]. The loop models a participant's key derivation similar to the key derivation used in PQ3. The inner loop applies a hash function repeatedly, using non-determinism to leave open how often, to a value derived from a KDF, which we call the *seed*. The outer loop updates the seed. When the outer loop starts, the seed is derived from a zero-byte sequence and a fresh value. At every outer-loop iteration, the seed is derived from the previous seed and either a zero-byte sequence or a fresh value (determined non-deterministically). The adversary can access all fresh values used in this loop using a reveal oracle.

The key derivation in this theory is similar to the key derivation in PQ3 when focussing on KEM shared secrets. The inner loop abstracts from the chain and message key derivation, while the outer loop abstracts from establishing new KEM shared secrets. At the end of this section, we show how a simpler version of this theory captures the essence of the double ratchet construction, i.e., repeatedly establishing fresh Diffie-Hellman shared secrets.

Now consider proving a simple key secrecy lemma: Every key established in the inner loop either remains confidential, or the most recently used fresh value in the outer loop was revealed to the adversary. To prove this lemma, we establish three auxiliary lemmas:

Outer Loop Step Every step in the inner loop must be preceded by a step in the outer loop. This lemma can be proven straightforwardly by induction.

Fresh Seed Source For every step in the outer loop ratchet, there must be a step in that ratchet deriving the seed that most recently was derived from a fresh value. We can prove this by induction using the *outer loop step* lemma. When proving this lemma, there is only one case that does not immediately lead to a contradiction. When the outer-loop step was immediately preceded by an inner-loop step, there is neither a contradiction nor does the induction hypothesis apply. In that case, we can apply the *outer loop step* to jump to the previous outer-loop step, which will either have used a fresh value to derive its seed (direct contradiction) or a zero-byte sequence (but since it is an outer-loop step, the induction hypothesis applies).

Seed Secrecy When the adversary derived a key established in the inner loop, they must have used the seed most recently derived using a fresh value in that derivation. This lemma can also be proven by induction straightforwardly.

Using these three auxiliary lemmas, Tamarin automatically proves key secrecy. Note that all auxiliary lemmas above are proven using induction but the key secrecy lemma is not.

We can describe above lemmas in more general terms.

Outer Loop Step This lemma "jumps to" the most recent step of the outer loop and allows one to skip unrolling the inner loop infinitely.

Fresh Seed Source This lemma "jumps to" the step in the outer loop that introduces the "relevant term" (in our case, the fresh term used instead of the zero-byte sequence). It allows one to skip unrolling the outer loop infinitely.

Seed Secrecy This lemma links the adversary's knowledge of the final key to adversary's knowledge of the "relevant term," i.e., the seed most recently established using a fresh term. It can be proven by induction using the previous two lemmas.

Note that a lemma that "jumps to" the relevant term in the outer loop (here *fresh seed source*) is only required when there can be unboundedly many outer-loop steps until the relevant step is reached (the relevant step being the one where the fresh term is introduced). To illustrate this point, we also provide a second nested loop theory (`nested-loop-simple` in [10]) that always uses a fresh value to establish the respective seed in the outer loop. In this theory, the lemma *fresh seed source* is not needed as unrolling the outer loop is sufficient.

This simplified theory is similar to the key derivation of PQ3 when focussing on the Diffie-Hellman shared secrets and ignoring the KEM shared secrets. It is also similar to the

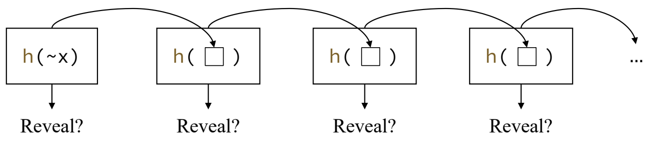


Figure 13: Revealing loop example.

double ratchet as used in Signal [3]. This simplified theory suggests that although the double ratchet construction used in Signal employs a nested loop, no inductive properties must be proven about the outer loop.

Revealing Loop Example The `revealing-loop` theory provided in [10] illustrates the challenges of proving properties of PQ3 when considering the reveal of root, chain, and message keys explicitly and thus illustrates how we addressed the problem of ghost sessions. This example is much simpler than the previous ones and is illustrated in Figure 13. A participant starts a loop in which they repeatedly apply a hash function `h` to some initial seed `~x`. Critically, the model allows the adversary to access any derived value and `~x` using a reveal oracle.

Again, we show how to prove a simple key secrecy lemma: Every hash value derived is secret, i.e., not known by the adversary, unless the adversary compromised any of the previous, intermediate values. To prove this lemma, we require two auxiliary lemmas:

Loop Start Every loop that computes a hash based on some initial seed `~x` is started, sampling `~x`. This lemma is straightforward to prove by induction.

Seeds Match If the computed hashes of two loops are identical, their seed must be identical too. Conceptually, this lemma is again simple and can be proven straightforwardly using induction.

With both these lemmas, TAMARIN can prove key secrecy of this example theory using induction. The two auxiliary lemmas help TAMARIN address the case that the adversary learns the hash in question from a ghost session. Using the second lemma, TAMARIN can connect the two sessions using a shared, fresh term. Then, using the first lemma, TAMARIN can instantiate the start of the loop where this shared, fresh term was sampled. From that, TAMARIN can deduce that both sessions must be the same; this enables it to apply the induction hypothesis and to prove the key secrecy lemma.

Again, we can generalize these auxiliary lemmas.

Loop Start This lemma is conceptually similar to the *outer loop step* lemma from the previous example and introduces no new kinds of lemmas.

Seeds Match This lemma links the computation of a value in a loop to the inputs to this computation, not determined by a loop (here, the seed).

Summary With the previous two examples, we showed how to handle nested loops and ghost sessions in TAMARIN. All auxiliary lemmas of these two theories match the three types of lemmas introduced in Section 5.4, which we briefly recapitulate:

Loop-Jump Lemmas These lemmas allow one to skip unrolling the steps of a (nested) loop. Examples: *outer loop step*, *fresh seed source*, *loop start*.

Adversary-Construction Lemmas These lemmas establish that for the adversary to construct one term, they must use another term. Example: *seed secrecy*.

Variable-Linking Lemmas These lemmas establish that for two instances of the same fact using two variables a and b, if both facts have the same value for a, they must have the same value for b. Example: *seeds match*.

A.3 Proof Methodology in General

Our proof methodology applies to theories that (i) use asymmetric cryptography to establish shared secrets, which in turn are used to derive symmetric encryption keys, (ii) include a (nested) loop computing these symmetric encryption keys, and (iii) for which trace formulas are to be proven of the following form:

$$\forall \vec{x}.C(\vec{x}) \implies (\neg)\exists \vec{y}.P(\vec{x};\vec{y}) \vee T(\vec{x}).$$

For all traces that satisfy some context C, there exists (or does not exist) an instance of P bound to that context (';' denotes vector concatenation), unless T (which specifies the threat model) applies. For example, for secrecy, C could be "a message was sent", P could be "the adversary learned that message" (in this case, non-existence would be proven), and T could be "the message encryption keys were revealed to the adversary."

We require that the protocol is modelled such that there is a single fact that models the protocol's (nested) loop, which we call the *loop fact*. This allows us to do two things: (a) exploit TAMARIN's heuristics for injective facts (see Section A.1.2), (b) clearly identify and relate looping variables, which will be critical to our proof methodology. We identify the loop fact's variables by their position in the fact, and there will generally be two kinds of variables: shared and derived secrets and key material used for establishing the shared and derived secrets. We relate the shared and derived secrets by a strict partial order. That order is defined as the smallest order closed under transitivity for which one variable a is smaller than another variable b if there is a state-transition rule that updates b using a. We say that a loop fact variable can *grow unboundedly* if there is no bound on the size of the terms that the variable can be unified with, for all ground-instantiated traces.

For example, our PQ3 model uses a `Session` fact to model the double-ratchet steps performed by a participant. The order on the shared and derived secrets is depicted in Figure 9 (the black arrows). For PQ3, the variables `msgKey`, `chainKey`, and `rootKey` can grow unboundedly.

Overall, our proof methodology has four steps.

1. For each of the loop fact's (possibly nested) loops, write a loop-jump lemma that connects a loop instance to its beginning, i.e., to the beginning of the loop overall or to the transition from an outer loop to the respective next inner loop.

2. Identify all the loop fact's shared and derived secret variables that can grow unboundedly (e.g., `msgKey` for PQ3). For each of these variables, write a variable-linking lemma that connects them to the variables that are directly smaller than them (e.g., a message to a chain key). For some of these unboundedly growing variables, it might additionally be necessary to write a loop-jump lemma that jumps to the rule application assigning a new value to the respective smaller variable. In our experience, this is the case for variables that are updated non-determinstically (i.e., `kemSS` for PQ3).

3. Identify all the loop fact's shared and derived secret variables that do not grow unboundedly. Typically, these variables will store shared secrets established using asymmetric cryptography. For each of these variables, write loop-jump lemmas that link the usage of that variable to the instantiation of the respective asymmetric key material used to establish the secret. The details of these loop-jump lemmas depend on the protocol specification. For example, the lemmas that link the ECDH shared secret to the respective ECDH keys substantially differ from the lemmas that link the KEM shared secret to the respective encapsulation key.

4. Finally, for all variables connected by variable-linking lemmas, write an adversary-construction lemma that states that in order for the adversary to know the contents of the respective larger variable, they must have either violated the threat model or know the respective smaller variables.

Following these steps, one would write the lemmas `RootKeyConnectionReceive`, `RootKeyConnectionSend`, and `SessionStart` in Step 1, the lemmas depicted in Figure 9 in Step 2, the lemmas depicted in Figure 10 in Step 3, and the lemmas depicted in Figure 8 in Step 4.

B Equational Theory for Protocol Model

We use TAMARIN's built-in equational theories for signing, symmetric encryption, and Diffie-Hellman key exchange. These respectively model digital signatures, symmetric encryption under message keys, and ECDH key exchanges. We additionally use TAMARIN's natural numbers theory to model message counters.

```
functions: hkdf/2, suffix/1, prefix/1, concat/2, h/1

equations: concat(prefix(x), suffix(x)) = x

functions: pqpk/1, encap/2, decap/2
equations: decap(encap(k, pqpk(sk)), sk) = k

functions: default/2, Just/1, None/0, unjust/1
equations: default(Just(v), t) = v,
           default(None, v) = v,
           unjust(Just(t)) = t
```

Figure 14: Custom functions and equations defined in our formal model.

In addition to these built-in theories, we specify some custom functions and equations, shown in Figure 14. First, we specify the functions `hkdf`, `suffix`, and `prefix` for key derivation. The function `hkdf` models an HMAC-based key derivation function [36] and takes two arguments: the first is the source of entropy and the second is a domain-separating tag or salt. The `prefix` and `suffix` functions are used for chain and root key derivations, which are derived by splitting a bit-string into a prefix and suffix of equal length. The function `concat` allows one to recover a value given its prefix and suffix. We do not need to use `concat` in the rules modeling the protocol roles of regular parties in our model, but the adversary can use it to reconstruct a value from the prefix and suffix. Additionally, we specify the unary function `h` to model the pre-key hash used during session establishment, see Section 4.

The functions `pqpk`, `encap`, and `decap` model KEM encapsulation and follow the standard symbolic model for asymmetric encryption. Finally, we use the wrapper function `Just` and the constant `None` to model optional values. The function `default` (together with the accompanying equations) unpacks an optional value or replaces it with a default. For example, we use `Just` and `None` to wrap values that are only sent optionally, e.g., the pre-key hash. The function `unjust` allows the adversary to access the contents of any `Just` value they intercept.

C PQ3 Messaging Protocol Description

In this section, we describe PQ3 in detail using pseudocode. We base our description upon technical material that we received from Apple researchers.

In our formal model, we only abstract from the pseudocode as presented here in that we operate in the symbolic model. For example, the root key derivation is actually done using two extract and one expand calls (see `rootAndChainKey` later). In the symbolic model, however, it only matters which entropy sources and domain separators are used to derive a value. Additionally, as the extract and expand operations are done by each party locally and without any interleaving, we faithfully model root-key derivation using one application of the `hkdf` function (see Appendix B).

We exclude the session handling layer, whose description was not available to us. Thus, we will leave two functions in our pseudocode undefined: `lookupSession` and `storeSession`. The effect of these functions should be clear from context, though. Our pseudocode follows Python syntax, and the type-definitions follow TypeScript conventions. We begin by defining some types for session handling. Variables that are followed by a `?` are optional and might have the value `None`.

```
type Session = {
  me: DeviceInfo,
  peer: DeviceInfo,
  inSenderRole: bool,
  messageIndex: uint32,
  expectKemSS: bool,
  rk?: byte[],
  ck?: byte[],
  ecdhKey?: P256.Key,
  newKemKey?: MLKEM768.Key,
  myKemKey?: MLKEM768.Key,
}

type DeviceInfo = {
  client_uri: byte[],
  device_id: byte[],
  ltk: P256.Key,
}

type PreKeyBundle = {
  ecdhKey: P256.Key,
  kemKey: MLKEM1076.Key,
  signature: byte[],
}

type Message = {
  ciphertext: byte[],
  authData: byte[],
  signature: byte[],
  ecdhPK: P256.Key,
  kemPK?: KEM.Key,
  kemEncap?: byte[],
  messageIndex: uint32,
  preKeyHash?: byte[],
  msgKeyIndicator: byte[],
}
```

We next turn to sending messages. The `sendMessage` function is called with the message most recently received (optional), and the participants' device information. It returns a message to be sent by the session handling layer. Where necessary, it may create a new session or perform the public-key ratchet.

```
def sendMessage(
  lastRcvd: Message | None, msg: string,
  me: DeviceInfo, peer: DeviceInfo,
) -> Message:
  sess = lookupSession(me, peer, lastRcvd)
  preKeyHash = None
  if sess is None:
    pkBundle = getPreKeyBundle(peer)
    sess = sessionStartSender(me, peer, pkBundle)
    storeSession(me, peer, sess)
    preKeyHash = SHA384(
        repr(pkBundle.ecdhKey)
      + repr(pkBundle.kemKey)
      + pkBundle.signature
    )

  encapResult = None
  if not sess.inSenderRole:
    encapResult, newKemKey
      = pkRatchetToSender(lastRcvd, sess)

  mk = symmetricRatchet(sess)
  ciphertext, msgKeyIndicator = encrypt(msg, mk)

  signature_body =
      b'messageSignature'
    + b'v1'
    + len(ciphertext) + ciphertext
    + msgKeyIndicator
    + repr(sess.ecdhKey.publicKey())
    + sess.messageIndex
    + dstForSession(me, peer)
    + len(newKemKey)
    + newKemKey.publicKey()
      if newKemKey is not None else b''
    + len(encapResult)
    + encapResult
      if encapResult is not None else b''
    + preKeyHash
      if preKeyHash is not None else b''

  return {
    ciphertext: ciphertext,
    signature: P256.sign(signature_body, me.ltk),
    ecdhPK: sess.ecdhKey.publicKey(),
    kemPK:  None if newKemKey is None
            else newKemKey.publicKey(),
    kemEncap: encapResult,
    messageIndex: sess.messageIndex++,
    preKeyHash: preKeyHash,
    msgKeyIndicator: msgKeyIndicator,
  }
```

The `receiveMsg` function performs the operations analogous to `sendMsg`. It starts with a signature verification, and proceeds to initialize a new session or perform the public-key ratchet, before it finally decrypts the message received.

```
def receiveMsg(
  msg: Message, me: DeviceInfo, peer: DeviceInfo,
) -> byte[]:
  signature_body =
      b'messageSignature'
    + b'v1'
    + len(msg.ciphertext) + msg.ciphertext
    + len(msg.authData) + msg.authData
    + msg.msgKeyIndicator
    + msg.ecdhPK
    + msg.messageIndex
    + dstForSession(peer, me)
    + len(msg.kemPK) + msg.kemPK
    + len(msg.kemEncap) + msg.kemEncap
    + msg.preKeyHash
  P256.verify_signature(
    signature_body,
    msg.signature,
    peer.ltk,
  )

  sess = lookupSession(me, peer, msg)
  if sess is None:
    sess = sessionStartReceiver(msg, peer, me)
    storeSession(me, peer, sess)

  if sess.inSenderRole:
    pkRatchetToReceiver(msg, sess)

  mk = symmetricRatchet(sess)
  return decrypt(msg, mk)
```

The next two functions specify the operations performed upon session start.

```
def sessionStartSender(
  sender: DeviceInfo, receiver: DeviceInfo,
  pkBundle: PreKeyBundle,
) -> Session:
  session = {
    me: sender,
    peer: receiver,
    inSenderRole: True,
    messageIndex: 0,
    expectKemSS: False,
  }

  ecdhKey = P256.new_key()
  ecdhSS =
    ecdhKey.shared_secret_with(pkBundle.ecdhKey)
  kemSS, encapResult = pkBundle.kemKey.encap()
  dst = dstForSession(me, peer)
    + b'session_start'
    + repr(pkBundle.ecdhKey)
    + repr(ecdhKey.publicKey())
    + repr(encapResult)
    + repr(pkBundle.kemKey.publicKey())

  rk, ck =
    rootAndChainKey(b'0' * 32, ecdhSS, kemSS, dst)

  session.ecdhKey = ecdhKey
  session.rk = rk
  session.ck = ck

  return session

def sessionStartReceiver(
  msg: Message, sender: DeviceInfo,
  receiver: DeviceInfo,
) -> Session:
  session = {
    me: receiver,
    peer: sender,
    inSenderRole: False,
    messageIndex: 0,
    expectKemSS: False,
  }

  bundle = getPreKeyBundleFromHash(msg.preKeyHash)
  ecdhSS =
    bundle.ecdhKey.shared_secret_with(msg.ecdhPK)
  kemSS =
    bundle.kemKey.decapsulate(msg.kemEncap)
  dst = dstForSession(sender, receiver)
    + b'session_start'
    + repr(pkBundle.ecdhKey.publicKey())
    + repr(msg.ecdhPk)
    + repr(msg.kemEncap)
    + repr(pkBundle.kemKey.publicKey())

  rk, ck =
    rootAndChainKey(b'0' * 32, ecdhSS, kemSS, dst)

  session.rk = rk
  session.ck = ck

  return session
```

The next two functions specify the operations performed in the public-key ratchet.

```
def pkRatchetToSender(
  msg: Message, sess: Session,
) -> byte[], KEM.Key:
  ecdhKey = P256.new_key()
  dst = dstForSession(sess.me, sess.peer)
    + b'pk_ratchet'
    + repr(msg.ecdhPK)
    + repr(ecdhKey.publicKey())

  ecdhSS = ecdhKey.shared_secret_with(msg.ecdhPK)
  kemSS = None
  encapResult = None
  if sess.newKemKey is not None:
    kemSS, encapResult = msg.kemKey.Encap()
    dst += encapResult
        + repr(msg.kemPK)
    sess.newKemKey = None

  newKemKey = None
  if heuristic():
    newKemKey = KEM.Generate()
    sess.myKemKey = newKemKey
    sess.expectKemSS = True

  rk, ck =
    rootAndChainKey(sess.rk, ecdhSS, kemSS, dst),

  sess.ecdhKey = ecdhKey
  sess.rk = rk
  sess.ck = ck
  sess.inSenderRole = True

  return encapResult, newKemKey

def pkRatchetToReceiver(msg: Message, sess):
  dst = dstForSession(sess.peer, sess.me)
    + b'pk_ratchet'
    + repr(ecdhKey.publicKey())
    + repr(msg.ecdhPK)

  ecdhSS = ecdhKey.shared_secret_with(msg.ecdhPK)
  kemSS = None
  if msg.kemEncap is not None:
    kemSS =
      sess.myKemKey.decapsulate(msg.kemEncap)
    dst += repr(msg.kemEncap)
      + repr(sess.myKemKey.publicKey())
    sess.expectKemSS = False
  elif sess.expectKemSS:
    raise Error()

  rk, ck =
    rootAndChainKey(rootKey, ecdhSS, kemSS, dst)

  if msg.kemPK is not None:
    sess.newKemKey = msg.kemPK

  sess.rk = rk
  sess.ck = ck
  sess.inSenderRole = False
```

Finally, we define some auxiliary functions. For example, they describe how the root and initial chain keys are derived, or how domain-separating strings are computed.

```
def dstForSession(
  sender: DeviceInfo, receiver: DeviceInfo,
) -> byte[]:
  return sender.client_uri + sender.device_id
    + repr(sender.ltk) + receiver.client_uri
    + receiver.device_id + repr(receiver.ltk)

def rootAndChainKey(
  oldRk: byte[], ecdhSS: byte[],
  kemSS: byte[], dst: byte[],
) -> byte[]:
  extracted = HKDF.SHA384.extract(
    IKM: ecdhSS,
    salt: oldRk
  )
  extracted = HKDF.SHA384.extract(
    IKM: extracted,
    salt: kemSS
  )
  rkCK = HKDF.SHA384.expand(
    PRK: extracted,
    info: b'rkDerivation-' + dst,
    L: 64
  )

  rootKey = rkCK.prefix(32)
  chainKey = rkCK.suffix(32)
  return rootKey, chainKey

def symmetricRatchet(sess: Session) -> byte[]:
  mk = HKDF.SHA384.expand(
    PRK: sess.ck,
    info: b'msgKeyDerivation',
    L: 32,
  )

  sess.ck = HKDF.SHA384.expand(
    PRK: sess.ck,
    info: b'chainKeyDerivation',
    L: 32,
  )

  return mk

def encrypt(
  msg: string, msgKey: byte[],
) -> byte[]:
  expanded = HKDF.expand(
    PRK: msgKey,
    info: b'aes-ctr',
    L: 48
  )
  iv = expanded.prefix(16)
  key = expanded.suffix(32)
  msgKeyIndicator = HKDF.SHA.384.expand(
    PRK: msgKey,
    info: b'msg-key-ind',
    L: 32,
  )
  ciphertext = AES_CTR.encrypt(msg, key, iv)
  return ciphertext, msgKeyIndicator

def decrypt(
  ciphertext: byte[], msgKey: byte[],
) -> string:
  expanded = HKDF.expand(
    PRK: msgKey,
    info: b'aes-ctr',
    L: 48
  )
  iv = expanded.prefix(16)
  key = expanded.suffix(32)
  return AES_CTR.decrypt(ciphertext, key, iv)
```