AlphaProof Code: Verifying Signal's Security at the Machine-Code Level

Swarat Chaudhuri[1], **Andrew Ferraiuolo**[1], **Anton Kovsharov**[1], **Tej Chajed**[1], **Sam Schlesinger**[2], **Karn Seth**[2], **Jonathan Katz**[2], **Charles Sutton**[1] **and Pushmeet Kohli**[1]
[1]Google DeepMind, [2]Google

We introduce AlphaProof Code, a multi-agent AI framework for formal software specification and verification that we used to prove key security properties of `libsignal`, the Signal protocol's reference implementation, at the machine-code level. AlphaProof Code consists of a human-guided agent for writing Lean specifications of code, a formal semantics of AArch64 machine code, and a team of autonomous prover agents whose proofs are accepted only if Lean's kernel checks them against the approved specification. Applied to `libsignal`, our agents generated about 6.5 million lines of machine-checked Lean proof code, resting on about 26,000 lines of trusted specifications and AArch64 semantics. The proofs cover Signal's post-quantum PQXDH handshake and Triple Ratchet, establishing forward secrecy, authentication, post-compromise security, and handshake secrecy. They are the first proofs connecting protocol-level security guarantees to the unmodified code of a widely used cryptographic protocol implementation. The results suggest that AI can make formal verification, historically too expensive for most software, practical for real-world systems. Our Lean development is available at `https://google-deepmind.github.io/code-verification`.

1. Introduction

Ensuring that the software that powers our world is correct and secure is one of the longest-standing challenges in computing [98]. Today, assurance primarily comes from testing, bug finding, and code review. However, reactive bug finding has inherent limitations, even when applying advanced AI systems for cyber-defense [29, 10, 77] to find and fix bugs quickly. An alternative is *formal verification* [54, 60, 61], where one specifies the correctness and security properties that a software system must enforce and then mathematically proves that the system satisfies them. Such a proof rules out every bug or attack that would violate those properties.

For all of its potential benefits, formal verification is notoriously difficult. While there have been heroic efforts to verify important, complex systems, they come at a high cost. The formally verified seL4 microkernel—perhaps the biggest success story in the field—took twenty person-years to verify [66]; the CompCert verified C compiler [73] took six. Further, in these successful examples, the source code of the system was written specifically with verification in mind. Verification of pre-existing industrial-strength code, such as a production library as compiled by an optimizing compiler, is even harder, as the prover cannot modify the code to make the proof simpler. Such labor costs are untenable for all but the most mission-critical domains.

In this paper, we show that recent AI advances can drastically reduce these costs. We present AlphaProof Code (Figure 1), a framework in which a human user and AI agents collaboratively write Lean [45] specifications of real-world code, and, once the user has vetted a specification, a team of autonomous agents proves that the code satisfies it. We used the framework to verify key security properties of `libsignal`, the reference implementation of the Signal protocol [91, 74], at the machine-code level. Our results are a significant milestone towards a grand challenge launched in April 2026 [23], and achieving them would have required years of expert labor with earlier tools.

© 2026 Google. All rights reserved

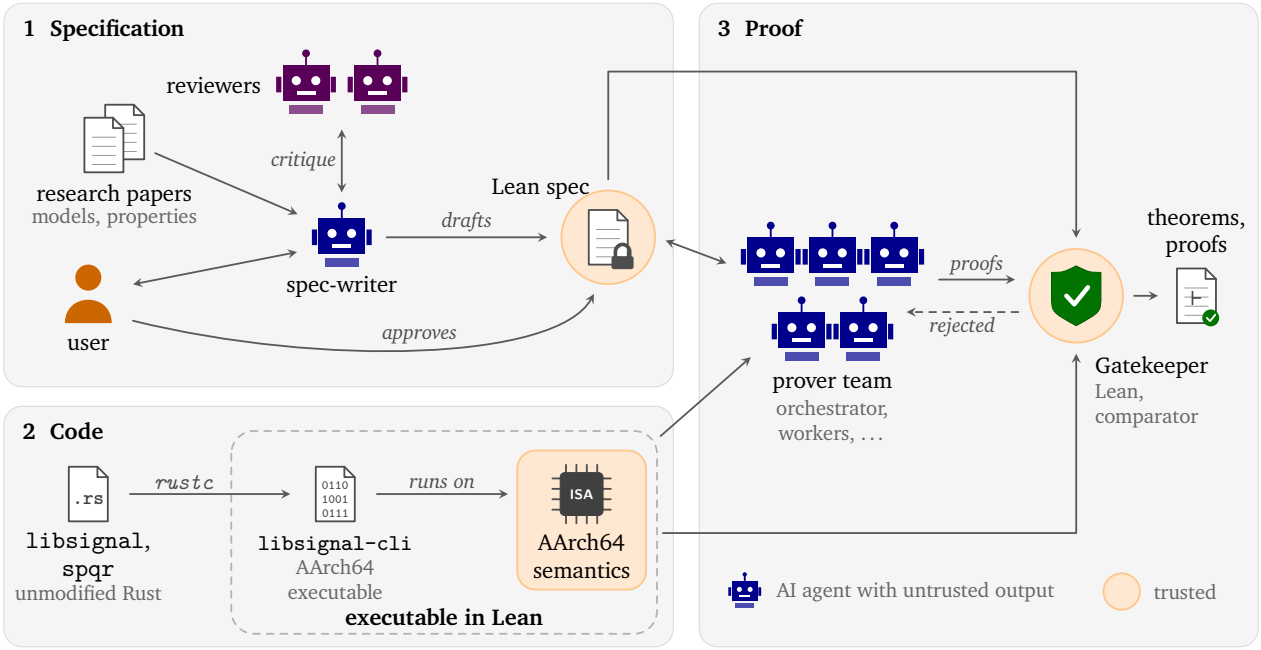


Figure 1 | The AlphaProof Code workflow. (1) A specification-writer agent works with the user to turn models and properties from the literature into a Lean specification. Reviewer agents critique each draft, and the user approves the result. (2) The unmodified Rust code is compiled with the standard compiler, and the executable is loaded into Lean together with a semantics of AArch64. (3) A team of prover agents proves the specification about the executable. If the specification is vacuous or otherwise incorrect, the provers can propose amendments, which are approved based on the user-set policy or explicit judgment. A gatekeeper agent accepts a proof only if Lean's kernel checks it and the Lean `comparator` confirms that it proves exactly a held-out copy of the specification. Shaded items are the ones a reader must trust. In addition, the results trust Lean's kernel, the security of the cryptographic primitives (stated as assumptions in the computational proof and idealized in the symbolic proof), and the contracts of some routines that the executable calls (§4).

Signal (§3) is the de facto global standard for end-to-end encrypted messaging. It protects the communications of billions of people worldwide: in addition to the standalone Signal application, variants of the protocol are deployed in WhatsApp, Meta's Messenger app, and Google Messages. The protocol includes sophisticated components such as the *PQXDH handshake* [69] and the *Triple Ratchet* [48], which are designed to ensure the security of messages even when some of the keys are compromised, and even against future adversaries with access to a quantum computer.

There is a long line of work on formal security verification of cryptographic protocols [49, 31, 17, 86, 15], including Signal [43, 28, 52, 40]. However, all of these efforts target abstract protocol models, verification-friendly reimplementations, implementations modified to simplify verification, or narrower properties such as crash-freedom (§8). By contrast, we give the first proofs of protocol-level security properties for a widely deployed, unmodified implementation of a cryptographic protocol. More generally, our work shows that AI can verify machine code against rich specifications from the research literature, including properties of the interaction between a stateful program and an adaptive adversary.

We link `libsignal` into a small command-line harness and compile it with the stock `rustc` compiler to AArch64 Linux machine code, with low-level cryptographic primitives specified by contracts (§4). We use AlphaProof Code to verify the code's security in the two standard models of protocol

security: the *symbolic* model (aka the Dolev–Yao model) [49], which treats those cryptographic primitives as abstract "black boxes" achieving standard security properties, and the *computational* model [56], which bounds an efficient attacker's success probability by the hardness of the underlying primitives.

Our specification-writing agents start with established symbolic [40, 28] and computational [7, 52] specifications for Signal and adapt them to Lean. However, the existing specifications do not take us all the way, as they analyze the handshake and the ratchet separately and predate Signal's post-quantum ratchet. With human guidance, our agents extend these specifications to the protocol as deployed. Our agent-written proofs of these specifications establish forward secrecy, post-compromise security, and message confidentiality and authenticity directly for the compiled binary (Table 1, §5.1, §5.2), against classical attackers, and, in the symbolic model, handshake secrecy against a future quantum attacker.

Our proofs amount to about 6.5 million lines of Lean. The statements and definitions a reader must read and trust are about 18,000 lines of code in our repository, plus about 8,000 lines of the AArch64 semantics and the symbol names in a Lean representation of the compiled machine code, whose bytes Lean checks against the executable. To check that our specifications catch real mistakes, we re-introduced two previously fixed `libsignal` bugs [40] and proved in Lean that the resulting binaries violate our symbolic security theorems (§6). Our Lean development, with all specifications and proofs, is available at the website `https://google-deepmind.github.io/code-verification`.

2. AlphaProof Code

AlphaProof Code (Figure 1) is a framework for human–AI collaboration on the formal specification and verification of real-world software. The framework is based on a simple principle: The majority of human effort in software verification now lies in engineering the *specification*, the formal statement that is proved and its assumptions about how the machine code executes. Previously, the expertise and labor required to write proofs was a bottleneck to large-scale formal verification, but this is no longer the case. As we demonstrate, even large proofs about software can be written by teams of AI agents once a specification is fixed, and the proof agents can work autonomously: since the proof is ultimately checked by the Lean kernel, the reliability of the results does not depend on the reliability of these proving agents.

For large, complex systems, AI assistance can reduce the cost of specification writing and assurance but not completely replace human judgment and review. Based on these observations, we propose that human authors select what to prove, resolve ambiguities, evaluate security trade-offs, and make all final decisions, while AI agents draft and test specifications under human direction.

We now describe the four components of AlphaProof Code (Figure 1) in turn.

2.1. Specification Writing

A *specification* in AlphaProof Code consists of a set of Lean theorem statements together with every definition they depend on, such as models of the code's environment. The specification defines what correctness and security mean for the program, and what classes of bugs and attacks the proof excludes. The specification must be *trusted*, meaning that the validity of the final result depends on getting it right, and it cannot be automatically checked—a flaw could weaken the guarantee the theorem provides (shaded in Figure 1). By contrast, the multi-million-line proofs produced by AI agents are *untrusted*: any logical error in a proof is rejected by Lean's kernel, preventing the proof from going through. The trusted definitions and theorem statements in the specification build on knowledge

available in research papers and standards. The definitions can also include extern contracts that define assumptions about parts of the system that are outside the scope of the verification campaign. For the Signal proofs, the trusted computing base is described in §5.3.

All agents in AlphaProof Code, including the specification writer, are general-purpose coding agents whose behavior is defined by *skills*: natural-language documents that describe an agent's task, the sources it should consult and the checks it must perform. The specification-writer agent drafts trusted theorems and definitions under close guidance from the user. A pool of reviewer agents checks that a draft is faithful to its sources and to the implementation.

Even when a model from the literature exists, it can take many rounds of iteration between the user and the agents to arrive at a good specification. For instance, prior formal models of Signal analyzed its initial handshake and its ratchet separately. Our specifications had to compose these models and cover Signal's new post-quantum ratchet (§5.1, §5.2), and this took several rounds of drafts and review. The user reviews, revises, and approves the specification before any proofs are written, and its git commit becomes the baseline that the gatekeeper later checks proofs against.

Much of this iteration is designed to ensure the specification's strength. A theorem that makes overly restrictive assumptions on the inputs, asserts only weak properties of the output, or places too few constraints on existentially quantified witnesses can hold even for buggy code. For instance, the computational security proof states that there exists a reduction from an attack on the protocol to a violation on an underlying assumption (§5.2.2). If a mistake in the specification weakens the capability of the attacker, then the security guarantee is also weakened. This is the sort of mistake that the iteration is designed to prevent.

For this reason, we use several techniques to test the specifications and assure their quality. Human authors review the specification; even so, specifications are like any large software artifact, in that human review is not expected to detect all bugs in a large codebase. Therefore, AlphaProof Code also includes several means of automated testing. First, reviewer agents are asked to provide reviews from a skeptical perspective, and in our experience were effective at detecting soundness and strength issues in the specification. Second, the prover agents create theorems to check the specification's quality, which are formally proved. For example, contradictory hypotheses make a theorem vacuous, so the agents also prove that the hypotheses are jointly satisfiable and that the theorems' premises are met by concrete executions, in the spirit of vacuity detection in model checking [20, 70]. Third, *mutation testing* in software engineering is a well-known technique for evaluating the effectiveness of test suites by injecting artificial mistakes into the code and seeing if the test suite fails [47, 64]. To test specifications this way, agents add natural bugs to the code, such as skipping the ciphertext MAC verification in Signal, and prove that the security theorem is false when applied to this mutated binary. Finally, we check that the specifications would have prohibited historical bugs in the system (§6), again in the sense that the symbolic security theorem is provably false.

2.2. Machine-Code Semantics

Our theorems are stated over a formal semantics of user-mode AArch64 Linux, written in Lean. The semantics takes an executable (an ELF binary) as a list of bytes and describes all of its possible runs as a relation between inputs, such as command-line arguments, standard input and the initial file-system state, and outcomes, such as the exit code, standard output and the final file-system state. The central definition, `ElfRunsTo bin pre post`, states that every run of `bin` from inputs satisfying `pre` ends in an outcome satisfying `post`, and that no run faults, for instance by reading unmapped memory. For the theorems to apply to real executions, the semantics must be *sound*: every behavior of the real hardware and kernel must be a possible run of the semantics. This cannot be proved, since

the hardware and kernel are not formal objects. Consequently, our approach is to keep the trusted definitions small and test them extensively.

The semantics models a single-threaded, statically linked process. A machine state holds the general-purpose and SIMD/floating-point registers, the program counter, the condition flags, a byte-addressed memory with per-page permissions, and the files in the current directory. A decoder turns the bytes at the program counter into an instruction, and the effect of each instruction follows Arm's machine-readable ASL specification [12, 13, 84]. Because code and data share one memory, writes to executable memory, accesses to unmapped memory and permission violations are faults, so a proof that a run terminates normally rules them out. Above the instruction level, a loader parses the ELF file, maps its segments and builds the initial stack, as Linux does when it starts a process. System calls are interpreted by a model of the calls the program uses: memory management, file and stream I/O, and exit. A run ends in one of two ways: it exits with an exit code, or it exits abnormally and faults due to an error, a panic, or an unmodeled instruction or system call. The semantics covers 467 instructions and 31 system calls in 7,532 lines of Lean. It does not model threads, signals, the vDSO, dynamic linking, or files outside the current directory, so programs that rely on them cannot be verified.

The trusted parts of the semantics, namely the decoder, instruction semantics, memory model, loader and system-call model, contain only definitions. The program logic used in the proofs is not trusted: its separation-logic rules, weakest-precondition calculus and proof automation are all proved sound in Lean against `ElfRunsTo`. We validate the trusted definitions by intensive testing. Instruction execution is tested in lockstep against two independent oracles: the Unicorn emulator (whose code is derived from QEMU), and the Sail model of Armv9.4-A [14], derived from Arm's official specification. The tests include single instructions, published test vectors, random instruction sequences and compiled Rust functions. The loader and the system-call model are tested against `qemu-aarch64` on a large number of binaries.

Two features of the semantics matter for specifications. First, the semantics treats the physical memory available to the process as a finite input, and a run that exhausts it ends in an out-of-memory outcome. A theorem then either proves, given a lower bound on available memory, that no run exhausts it, or accounts explicitly for the runs that do (§5.2.4). It is assumed that the operating system provides physical memory if available. Second, a proof can *hook* a function, which is then a stub in the executable: a call to it is modeled with a trusted contract instead of executing code. This is how assumptions about cryptographic primitives enter a result, since their security cannot itself be proved, and it is also used for a few routines such as the functions that write files and standard output. Hooks are part of the specification and are reviewed carefully (§2.1).

2.3. Proof Synthesis

Our proofs are produced autonomously by a team of agents with distinct roles, each defined by a skill. An *orchestrator* divides up the work and schedules agents but never edits Lean files itself. Each subtask is assigned to a *proof worker*. The main difficulty in a large team is managing context: each agent's context window is limited, and state can be lost when the context is compacted. The orchestrator therefore groups workers under *team leads*, each responsible for a larger, longer-running goal. A lead absorbs the back-and-forth with its workers and sends the orchestrator short summaries of open obligations, completed lemmas and blockers. Agents also try to falsify a goal before work on it starts, to avoid wasting resources on unprovable subgoals.

A few fixed roles keep the shared codebase coherent as the proofs grow. An *architect* designs, and periodically revises, the high-level decomposition of the proof. An *integrator* merges verified proofs

into library modules, removes duplicate lemmas and generalizes lemmas for reuse. To avoid conflicts among dozens of concurrent agents, each file is edited by at most one worker at a time, and only the integrator commits to the repository. Other support roles schedule expensive Lean builds and monitor agents and resources. In our proof campaigns, a team typically had 40 to 80 agents and ran on a single machine for 24 to 72 hours.

Proof attempts can reveal that a specification is subtly wrong about the code or its environment. Agents cannot edit the specification, but they can propose an *amendment* to a *critic* agent, which never sees the proof attempts. The critic accepts an amendment only if three conditions hold. First, the amendment must be necessary, as shown by a machine-checked disproof, a counterexample on the binary or an argument grounded in the code. Second, it must not weaken security, and any assumption it adds must hold of the real code. Third, it must be small, typically a few lines in one definition. Amendments that change trusted assumptions are approved by the user or under a user-set policy, and each is logged with its justification. Examples from our campaign include amendments that strengthen trusted contracts for library functions to match their real behavior, for example, "`memcpy` may copy from read-only memory."

2.4. Gatekeeping

A proof is accepted only if it proves exactly the statements the user approved. Inspecting the theorem statement alone is not enough to check this, since a statement can be weakened indirectly by changing a definition it depends on. The *gatekeeper* is an agent used during development to check recorded approvals for specification amendments (if any) and to call the standard `comparator` tool [72] to check the proof.

The agent invokes `comparator` using a deterministic script that can be rerun manually when desired. We keep a strict discipline of which files are in the trusted specification of the theorem versus the untrusted proof, enforced by deterministic scripts, which allows clean rerunning of `comparator`. At the start of a campaign, the gatekeeper copies the specification from the baseline commit into a *held-out* directory that no agent can write to. The held-out copy changes only through accepted amendments (§2.3), which are reviewed by the critic and approved by human authors before the gatekeeper applies them.

The gatekeeper accepts a proof only if `comparator` confirms that the statement and every definition it depends on are identical to this copy, that the proof uses only Lean's standard axioms, and that Lean's kernel accepts the proof. At that point, the gatekeeper declares that the proof is complete (proof agents, team leads, and the orchestrator cannot declare a top-level goal done on their own). When proof development is complete, we re-run `comparator` manually for further assurance.

3. Background on Signal

Signal is a protocol for asynchronous messaging, meaning that it allows a user Alice to start a conversation with a user Bob and send him messages while he is offline. A server stores and relays messages but is not trusted for either confidentiality or authenticity. We describe the version implemented by `libsignal` v0.105.4, restricted to one-to-one sessions. (We leave group messaging as future work.) A session has two phases (Figure 2): a *handshake*, in which Alice and Bob agree on a shared secret, and a *ratchet*, which derives a fresh key from that secret for every message.

The handshake relies on Diffie-Hellman (DH) key agreement: Alice combines her private key with Bob's public key, Bob does the reverse, and both get the same value, which an eavesdropper cannot compute from the public keys alone. Since Bob may be offline, he leaves public keys, called *prekeys*,

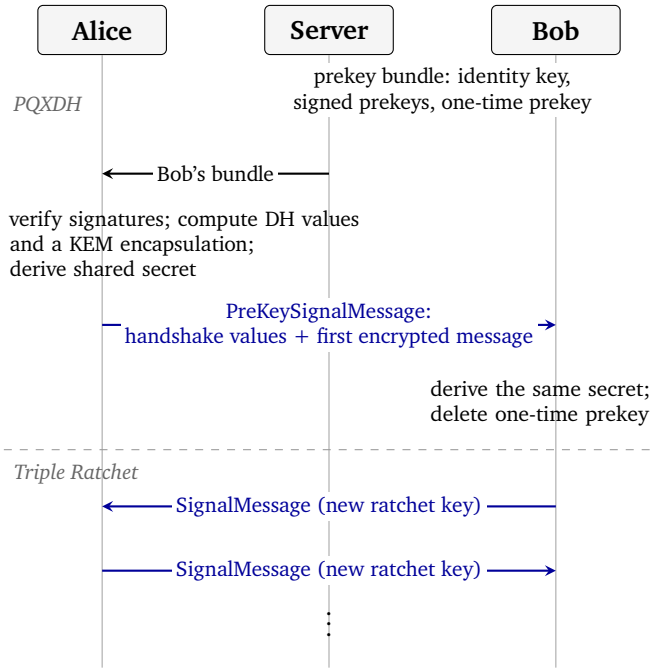


Figure 2 | A Signal session as implemented by `libsignal` [74, 91]. Bob uploads his prekey bundle to the server; Alice fetches it, completes the handshake (PQXDH) on her own, and sends a prekey message that carries the handshake values together with her first encrypted message. She keeps sending prekey messages until Bob replies; after that, the parties exchange ordinary Triple Ratchet messages, each under a fresh key. Figure 8 in Appendix A adds the notation and the key schedule.

on the server for senders to use. Signal's handshake, PQXDH [69], mixes several DH values with a post-quantum key encapsulation mechanism (KEM). The resulting secret survives the leak of some of these keys, and an eavesdropper who records the handshake cannot recover it even with a future quantum computer.

The ratchet, which Signal calls the Triple Ratchet [48], combines the classical Double Ratchet with the Sparse Post-Quantum Ratchet (SPQR) [89]. This design contains the damage from a stolen key. Each message key is derived from the previous state by a one-way function and then deleted; stealing a party's current state therefore exposes no earlier encrypted messages (*forward secrecy*). Every reply also mixes in a fresh DH value, which locks out an adversary who stole the state, as long as it stays passive during that fresh DH exchange (*post-compromise security*).

Our goal is to show that these properties hold against a strong adversary that controls the network and the server and may compromise long-term keys, prekeys or a party's entire state. For the post-quantum properties, the adversary may later break DH. §5.1 and §5.2 state secrecy, authentication, forward secrecy and post-compromise security precisely.

The subsequent sections of this paper refer to a few more of `libsignal`'s components. Bob's prekey bundle holds his long-term X25519 identity key, a *signed prekey* and a Kyber1024 prekey,[1] both signed with the identity key using XEdDSA, and optionally a *one-time prekey* that the server hands out only once. On the ratchet side, a *root key* absorbs the fresh DH values and yields *chain keys*, from which message keys are derived. SPQR runs a separate ML-KEM-768 exchange and mixes its keys into every message key, so that post-compromise security also holds against a quantum adversary. Appendix A has the full key schedule.

[1] Kyber1024 is the round-3 predecessor of ML-KEM-1024 (FIPS 203); `libsignal`'s PQXDH still uses it.

	Symbolic model (§5.1)	**Computational model** (§5.2)
Attacker	Controls the network and can compromise long-term keys, prekeys and session state. Can also break DH (modeling a quantum attacker), or the KEM (modeling cryptanalysis). Unbounded, but limited to symbolic operations on messages.	Controls the network and can compromise long-term keys, prekeys and session state, except at a few points where compromise would make winning trivial. Any classical algorithm whose running time and queries are bounded.
Guarantee	Ten theorems, each holding on every collision-free execution (Table 2): forward secrecy and post-compromise security of message keys, handshake authentication and KCI resistance, and handshake secrecy, even if a quantum attacker later breaks DH. No run faults or diverges.	Confidentiality and authenticity of messages, with forward secrecy and post-compromise security, for PQXDH composed with the Triple Ratchet, as a concrete bound on the attacker's advantage (Theorem 2).
Assumptions	Idealized primitives: hashes, KDFs and MACs never collide and cannot be inverted, secrets never repeat, and ciphertexts (resp., signatures) cannot be decrypted (resp., forged) without the key. These are idealized forms of standard security properties that real primitives are believed to have, except with negligible probability.	Gap-DH and XEdDSA unforgeability, Kyber1024 collision resistance, PRF/PRG security of HKDF and HMAC, and one-time CCA security of AES-CBC with HMAC (§5.2.4).
Trusted model	The symbolic model and the event layer that maps runs of the binary to symbolic-model events.	The formalized security definitions and assumptions, and the byte-level Lean model of `libsignal` (§5.3).
Not covered	Computational secrecy, more than two parties, and SPQR's post-quantum healing (§5.1.6).	Quantum attackers, multi-device state, adversarial first messages to sessions that are later promoted, and large scale: the bound becomes vacuous for multiple long sessions with many epochs (§5.2.5).

Table 1 | Summary of our verification results. We verify `libsignal-cli`, an AArch64 executable built by stock `rustc` from the unmodified `libsignal` v0.105.4 and SPQR crates. Cryptographic primitives, the two output functions and a few library routines are specified by contracts (§4); everything else is verified. Besides the specifications, the proofs require trust in Lean's kernel and the AArch64 semantics, though not the Rust source or compiler.

The code does much more than the protocol described above, and this is where implementation bugs arise. For example, `libsignal` keeps a large mutable session state, stores keys for messages that arrive out of order, splits SPQR's ML-KEM messages into chunks spread over ordinary messages, and uses protobuf for serialization. It also keeps archived earlier sessions so that delayed messages can still be decrypted. Because our theorems are about the compiled code, they cover this machinery. In one now-fixed bug, a prekey message that decrypted only under an archived session caused `libsignal` to store the new session's state anyway, including a copy of the signed prekey's secret. When we reintroduce it, the resulting binary provably violates our symbolic security theorem (§6).

Table 1 summarizes our verification results for the protocol. More details are available in §5 and the appendix.

4. Setup for Verification

We verify the one-to-one session code of `libsignal-protocol` (v0.105.4, with `spqr` v1.6.0): PQXDH session setup and Triple Ratchet messaging. Rather than reasoning about its Rust functions and in-memory state, we wrap the unmodified crates in a small command-line harness, `libsignal-cli`, and state our theorems about whole runs of its compiled AArch64 executable. Each run of this tool performs one operation for one party specified with command-line arguments: `keygen` creates the party's keys, `init` starts a session from a prekey bundle (`process_prekey_bundle`), and `send` and `recv` encrypt and decrypt one message (`message_encrypt`, `message_decrypt`). Our theorems cover `init`, `send` and `recv`; `keygen` is not verified, and the theorems instead take the parties' keys as part of their setup.

States. A party's state is stored in a directory (Figure 3 and Figure 4). Its files hold everything `libsignal` stores for the party, in `libsignal`'s own serialization format: its identity key pair, its sessions (one per peer device), its signed, one-time and Kyber1024 prekeys, and the identity keys of its peers. The subdirectory `public/` holds what a client uploads to the Signal server: the public halves of its keys and their signatures. Nothing else survives between runs, so leaking a party's directory is exactly a compromise of its whole state.

Process semantics. A run depends only on its command line, its standard input, the party's files, and a file `coins` that supplies all of its randomness; the clock is fixed and no other entropy source is read. A run produces only an exit code, its standard output, and new contents for the party's files. Exit code 0 means that `libsignal` accepted the operation: the changed files are written, and the output (a wire message for `send`, a plaintext for `recv`) is on standard output. Exit code 1 means that `libsignal` rejected it, and then *no file is written*: `libsignal` updates its stores one at a time and may fail after some have changed, so, like Signal's own mobile apps, `libsignal-cli` holds back all store writes until the operation succeeds. (Exit code 2 reports a malformed command line or an unreadable file.) Our theorems speak only about these inputs and outputs, using a Lean model of how Linux loads and runs an AArch64 executable.

Between runs, control passes to the *environment* (Figure 3). It runs every call of `libsignal-cli`, holds all of the parties' files, plays Signal's server by building each prekey bundle as a directory of copies of the peer's `public/` files, and supplies the `coins` of every run. The attacker lives inside the environment. It decides which party runs which operation, and when. It controls the network: the environment hands it every wire message that a run writes to its standard output, and it chooses what each run reads on its standard input, so it may drop, reorder, replay or modify messages. When it asks to compromise a party, the environment gives it that party's files. This picture is common to

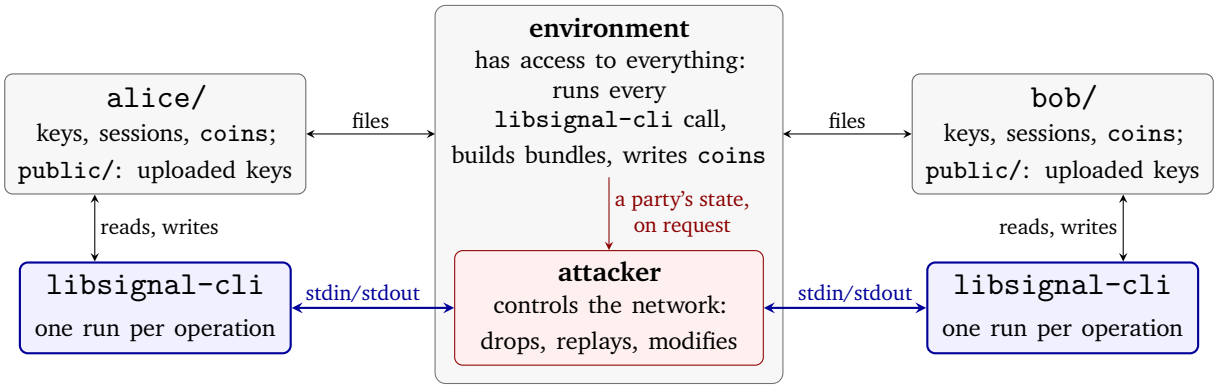


Figure 3 | Each party's state is a directory, and each operation is one run of the `libsignal-cli` executable in that directory. The environment has access to everything: it runs every call, plays the role of the Signal server by building prekey bundles from the parties' `public/` files, and writes the randomness of every run. The attacker, inside the environment, controls the network and gets a party's state when it asks to compromise it.

Bob after `keygen`

```
bob/
├ identity-key-pair
├ identities/  (empty)
├ sessions/    (empty)
├ signed-prekeys/1
├ one-time-prekeys/1            71 B
├ one-time-kyber-prekeys/2    4812 B
├ last-resort-kyber-prekeys/1
├ public/  (6 files: the keys Bob uploads)
└ coins
```

`recv $ALICE < m1` → exit 0

Bob after receiving `m1`

```
bob/
├ identity-key-pair
├ identities/<alice>            33 B
├ sessions/<alice>.1           639 B
├ signed-prekeys/1
├ one-time-prekeys/1             0 B
├ one-time-kyber-prekeys/2       0 B
├ last-resort-kyber-prekeys/1
├ public/  (unchanged)
└ coins
```

↻ `recv` of `m1` again: exit 1, no file written

Figure 4 | Bob's directory before and after he receives Alice's first message `m1` (abridged; sizes from a real run). The successful `recv` stores Alice's identity key and the new session (green) and empties the two one-time prekeys that `m1` used (red): files are emptied rather than deleted. Every other file is unchanged. Delivering `m1` a second time is rejected and writes no file.

```
ALICE=8a6f1c2e-3b4d-4e5f-9a0b-1c2d3e4f5a6b.1  BOB=5d3e2f10-7a8b-4c9d-8e0f-a1b2c3d4e5f6.1
run()    { head -c 16384 /dev/urandom > $1/coins; (cd $1 && libsignal-cli "${@:2}"); }
bundle() { mkdir $2; cp $1/public/⟨identity key, one prekey of each kind⟩ $2/; }
mkdir alice bob
run bob keygen $BOB 1                          # Bob's keys; public halves in bob/public/
run alice keygen $ALICE 0
bundle bob bob_public_bundle                   # play the server
run alice init $BOB ../bob_public_bundle       # PQXDH
printf "Hi Bob!" | run alice send $BOB > m1    # m1: a PreKeySignalMessage
run bob recv $ALICE < m1                       # prints Hi Bob!
printf "Hi Alice!" | run bob send $ALICE > r1  # r1: a SignalMessage
run alice recv $BOB < r1                       # prints Hi Alice!
run bob recv $ALICE < m1; echo $?              # replay: prints 1; no file written
```

Figure 5 | A typical scenario as a `bash` script, which plays both the environment and the attacker. `run` writes fresh `coins` into a party's directory and runs `libsignal-cli` there; `bundle` plays Signal's server, copying a party's public identity key and one public prekey of each kind into a new directory.

both of our proofs; they differ in how they formalize the environment and the attacker's capabilities. In the computational proof, the environment is a security game and the attacker is any probabilistic algorithm that makes a bounded number of queries; the security proof bounds its advantage at telling which of two challenge messages was encrypted or getting a forged message accepted. In the symbolic proof, the runs and the environment's actions are matched by transitions of the DyLean [99] model of the protocol, a compromise leaks the secrets in the party's files one at a time, and the attacker is a Dolev–Yao attacker that computes on symbolic terms; security is a property of every trace.

Because `libsignal-cli` is an ordinary executable, the same scenarios can also be run directly from a shell. The script in Figure 5 plays both the environment and the attacker: it creates Bob and Alice, plays the server to give Alice a bundle of Bob's keys, runs the PQXDH handshake and two Triple Ratchet messages, and finally, as the attacker, replays Alice's first message, which Bob rejects without writing any file. We also ran these scenarios on our Lean semantics of the binary (15.6 million instructions across 10 invocations), and every run had the same outcome as on the real binary, including all session state.

Verification target. We compile `libsignal-cli` and the unmodified `libsignal` crates with stock `rustc` (1.98.1) and link them statically with musl into one AArch64 Linux executable, in which each *hooked routine* (below) is a stub (build details in Appendix C.6). The proofs follow each `init`, `send` and `recv` run from the executable's entry point to its final `exit_group` system call. The proof must reason instruction by instruction about the CLI and its file-backed stores (including the routines that read files and standard input), musl's `mallocng` memory allocator, string cloning, the PQXDH, Triple Ratchet and SPQR state machines, key derivation (HKDF and `libsignal`'s HMAC-SHA256, apart from the hooked SHA-256 compression function), protobuf serialization and MAC verification. The remaining 39 hooked routines are replaced by contracts at the binary boundary, and 18 panic entry points end the run with `SIGABRT`. The hooked routines are the low-level cryptographic routines (SHA-256 and SHA-512 compression, AES-256-CBC, the X25519 and Edwards curve operations used by XEdDSA, Kyber1024 in PQXDH and ML-KEM-768 and HMAC from `libcrux` in SPQR); five `libc` memory and string routines (`memset`, `memmove`, `memcpy`, `bcmp` and `strlen`); the two functions that write files and standard output; and a few small Rust library routines: string formatting, UTF-8 checking, `memrchr`, the fixed clock, an optimization barrier, and the zeroizing destructors of X25519 secrets. In our main theorems, each hooked routine is a fixed Lean function; the cryptographic primitives inside these functions are left abstract. The functional-correctness theorems proved about the binary are shared by the computational and the symbolic security proofs.

The contracts for the cryptographic routines idealize the primitives, since our results are conditional on their security. The remaining contracts describe the routines' functional behavior. They are not proved against the real routines, but the scenarios of Figure 5 have the same outcomes when run with them in the Lean semantics as on the real binary.

While our setup models a specific execution environment, our results are relevant to many others. For example, the Signal apps for Android and iOS run the same Rust code for `libsignal`. Nearly all Android phones and all iPhones use AArch64; the difference is in compiler flags, linking, and the operating system interface. The theorems we prove carry over to these other execution environments as long as we trust that they preserve the behavior of the protocol code as exercised by the `libsignal-cli` binary.

4.1. Binary-Level Guarantees

The security proofs described in this paper build on a set of theorems about the machine code. These theorems are proved once and for all. They hold for every placement of the code and of musl's allocator in memory that matches the bytes of the executable, and Lean checks that such placements exist. They also hold for every implementation of the hooked routines.

These theorems state that the executable implements the Lean models of `libsignal` on which the security proofs are built. For example, our computational security theorem (through Lemma 4) states that on every `init`, `send` and `recv` run whose inputs are within the stated size limits, the executable does what the model prescribes. If the model accepts the operation, the run exits with code 0, prints the model's output, and leaves exactly the model's new state in the party's files. If the model rejects it, the run exits with code 1, prints nothing, and leaves every file unchanged. If the model panics, the run ends with `SIGABRT`. The CLI's own checks are proved as well: a run exits with code 2 if its standard input exceeds 2^{20} bytes, and `send` rejects plaintexts longer than 10^6 bytes. In particular, no run faults, loops forever, or leaves a state that the model would not produce. The symbolic refinement proof is built on the same facts.

The only exception these theorems allow is a run that exhausts the machine's memory. A separate theorem shows that this does not happen: in our process model, which gives each run 1 GiB of memory beyond its loaded segments and an 8 MiB stack, no run on inputs within the limits reaches the out-of-memory state. Its proof bounds the heap use of every run through the verified allocator, using the fact that a session record stays below 1.2 MB however many messages the session carries. The security theorems therefore contain no out-of-memory term, and the symbolic theorem states outright that every valid call exits or aborts and never faults or diverges.

These guarantees concern a fresh process started with only standard input, output and error open, an empty environment, and a fixed clock. The kernel's random bytes are fixed, so all of a run's randomness comes from its `coins` file. The hooked routines are replaced by their contracts. The guarantees say nothing about a machine with less memory, about processes launched with other files open, or about the code of the hooked routines themselves. The security theorems constrain those routines separately.

5. Verifying Signal

We verify the security of the `libsignal-cli` binary (§4) in two established models from the research literature: the *symbolic* model (§5.1) and the *computational* model (§5.2). In the symbolic model, cryptographic primitives are idealized and output abstract terms. The goal is to show that in every possible execution, secret values (e.g., messages) remain secret and parties agree on whom they are talking to. In the computational model, messages are bit strings and the adversary is a probabilistic algorithm that interacts with the protocol through a bounded number of oracle queries. The proof objective is to show that breaking the protocol is at least as hard as breaking the cryptographic primitives it is built from: any adversary's advantage in attacking the protocol is bounded by a function of the advantages of efficient reductions that break some underlying cryptographic primitive.

We work in two independent models for two reasons. First, we wanted to learn whether agents can carry out proofs in both models, which differ in their tools, their methods of reasoning and the ways they fail. Second, the two proofs give different assurances about the protocol. The computational proof bounds the advantage of any query-bounded adversary and accounts for events, such as collisions, that the symbolic proof abstracts away. The symbolic approach, in turn, follows `libsignal`'s session records more closely and states ten properties under a richer set of compromises. The symbolic proof

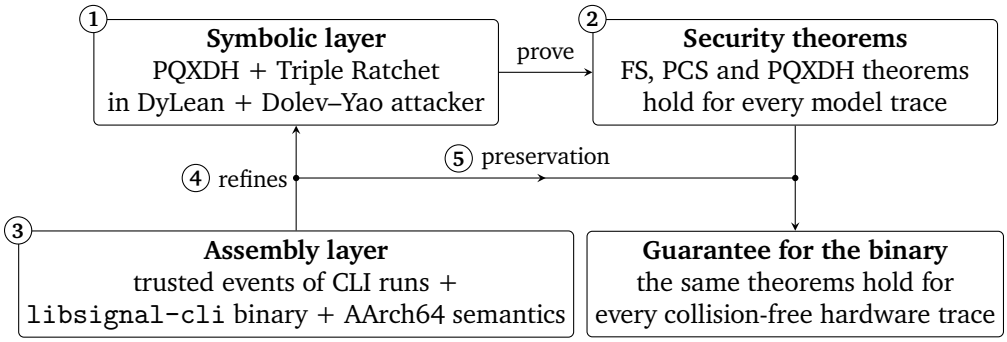


Figure 6 | Overview of the symbolic proofs about `libsignal`. Refinement shows that every hardware trace of the compiled binary is explained by a model trace, up to collisions, so preservation carries the model's theorems (Table 2) over to the binary. Circled numbers match the steps listed above.

also covers an attacker (for example, a future quantum computer) that breaks Diffie–Hellman after the handshake. By contrast, our computational proof only covers classical adversaries.

5.1. Symbolic Proofs

We prove that the `libsignal` binary enforces symbolic security properties of the composition of PQXDH and the Triple Ratchet, down to the assembly level. The symbolic model of cryptography simplifies verifying the high-level design of cryptographic protocols by assuming the underlying primitives obey "ideal cryptography" (e.g., no hash collisions, encrypted values are meaningless to attackers without the key). This enables a separation of concerns: developers of the underlying primitives can validate these standard assumptions, while a symbolic analysis of a protocol can rely on them to prove protocol-specific properties.

Most symbolic security verification efforts verify abstract designs and give no assurance about the real implementation. We instead define and prove symbolic security properties of the compiled binary using the following approach:

1. *Symbolic Model.* We use DyLean [99], an existing Lean framework for symbolic cryptography, to write one composed model of PQXDH and the Triple Ratchet (§5.1.1).
2. *Security Theorems.* We state and prove that the model satisfies ten security properties, replicating the symbolic state-of-the-art for Signal [40, 28] (§5.1.2).
3. *Assembly-level Symbolic Events.* To state symbolic security at the binary level, a thin trusted event layer turns executions of the binary under our AArch64 semantics into *hardware traces* of protocol events: one event per run of `libsignal-cli`, and one per leak, break or inversion by the attacker between runs (§5.1.3).
4. *Refinement.* We prove that every hardware trace is explained by a model trace, up to collisions. A trusted concretization function γ gives the bytes that each model term stands for (§5.1.4).
5. *Preservation.* We prove that refinement and the model-level theorems together imply the same ten properties, for an attacker that knows only what it observed of the execution, on every hardware trace whose coins never repeat a secret: on the whole trace if it is collision-free, and otherwise on its explained prefix before the collision (i.e., our refinement definition preserves symbolic security; Theorem 1, §5.1.5).

The first three items comprise the trusted definitions (§5.3); everything that connects them (the model-level proofs, refinement and preservation) is proved in Lean.

Figure 6 shows an overview of this approach. Steps 1 and 2 follow prior work closely. The

challenges are in steps 3 and 4, because the binary does not satisfy the symbolic assumptions. First, hardware has no messages: the CPU only reads and writes registers and memory. We address this with a small, trusted augmented program semantics that primarily calls our AArch64 one, but extends it with these events as described in §5.1.3. Second, hardware values are finite byte strings, not terms. A Dolev–Yao attacker cannot guess a key, but an attacker of the binary can guess `libsignal`'s 8-byte MAC with probability 2^{-64}, so the binary does not refine the model exactly. Following Arquint et al. [15], §5.1.4 maps model terms to bytes but never maps bytes back to terms. A guessed value is then a different term whose bytes happen to collide.

Crucially, we do not aim to advance the state of the art in what has been proven about the Signal protocol [74, 91] at the symbolic level. Our contribution is replicating the state of the art at the symbolic level and rigorously connecting it to the assembly implementation for the first time.

5.1.1. Symbolic Model

DyLean [99] is a Lean framework for symbolic protocol analysis in the style of ProVerif [33] and Tamarin [76]. A protocol is a labeled transition system that produces atomic transitions over a global append-only trace. The transitions are scheduled by the (Dolev–Yao) attacker, who also chooses their inputs. Security theorems are described in terms of the attacker's *knowledge* of a trace, which is the least set of terms that contains everything the trace reveals (its network messages, including leaked secrets) and is closed under attacker operations of each primitive. The attacker may inject only terms it knows, so it cannot guess a key or forge a signature.

Attacker operations and symbolic assumptions are both modeled through *equational theories*, which model the underlying primitives through trusted (assumed) rewrite rules. For instance, `dh` sorts its two secret exponents, so `dh(dhPk(a), b)` and `dh(dhPk(b), a)` are the same term, and `symDec k iv c` returns `m` exactly when `c` is `symEnc k iv m` and fails otherwise.

The composed model. Using DyLean, we formalize a single composed symbolic model of the deployed protocol by combining the PQXDH handshake model of Bhargavan et al. [28] with the Triple Ratchet model of Cheval, Jacomme, and Richards [40]. Because the binary must refine it, the model is more faithful to `libsignal` than the papers' models in two places: it models X25519's non-canonical encodings and small subgroup, which are outside the scope of Cheval et al.'s proofs and are the setting of the small-subgroup bug of §6; and it models `libsignal`'s binding of the KEM public key into the shared secret, rather than the change to the associated data (PQXDH Revision 2) that Bhargavan et al.'s ProVerif proofs assume.

The composed model is a labeled state machine covering initial prekey bundle processing (with signed and one-time X25519 prekeys and reusable, last-resort Kyber prekeys; one-time Kyber prekeys are not modeled), prekey message reception, and subsequent Triple Ratchet messaging (the Double Ratchet coupled with SPQR), with unboundedly many sessions but at most one active session per party and role, scheduled by a Dolev–Yao attacker that controls the network and may compromise long-term keys, prekeys, or stored session state. Following Bhargavan et al. [28], the attacker may also break DH (say, with a quantum computer) or KEM (say, with cryptanalysis) and from then on invert public keys of the broken primitive to learn their secrets. Appendix B.1 describes the model as a transition system.

5.1.2. Security Theorems

Table 2 lists the theorems we prove. They restate the forward secrecy and post-compromise security properties that Cheval et al. [40] prove for the Double Ratchet after PQXDH, and the PQXDH theorems

of Bhargavan et al. [28]; both papers use ProVerif [33]. All of them hold on every reachable trace of the composed model described above. Neither paper analyzes exactly this protocol, so the last column of the table gives the design-level differences between each paper's statement and ours. We use `libsignal`'s names and roles: Alice processes Bob's prekey bundle and sends the first message, so she is the initiator and Bob the responder. The PQXDH paper swaps the two labels: our initiator secrecy is its responder secrecy (Theorem 7), and our responder secrecy its initiator secrecy (Theorem 9).

Most of the differences are forced by verifying the deployed code rather than a protocol description: `libsignal`'s three handshake outputs instead of a single key, KEM public-key binding in the shared secret instead of the associated data, and agreement on prekeys only up to re-encoding. To rule out vacuous theorems, we also prove that honest sessions with and without a one-time prekey are reachable, and that the premises of the forward secrecy and handshake secrecy theorems can be met. At the binary level, we prove that the initial setup and four example interactions (handshakes with and without a one-time prekey, an honest session, and one with a leaked key) arise from runs of any implementation that meets the CLI's functional specification, and that these runs meet every hypothesis of Theorem 1 without a collision.

5.1.3. Assembly-Layer Events

To state security at the assembly level, we add protocol events as a thin, trusted layer on top of the unchanged AArch64 semantics. Protocol-level events are naturally observed at the process interface of `libsignal-cli` (§4): its command line, standard streams and files already fix which bytes are a plaintext, a wire message or randomness, so no event needs instrumentation inside the binary. A *hardware trace* is a list of events. Each records either one run of `libsignal-cli` (`init`, `send` or `recv`) as the environment sees it, or an action of the attacker between runs: leaking a secret from a party's directory, compromising a party's DH or KEM, or inverting a public key of an honest party after the matching compromise. The environment may also delete a party's session file between runs; this emits no event and is how Alice starts a new session. Appendix B.2 lists every event (Table 4).

5.1.4. Refinement

Refinement relates hardware traces to traces of the DyLean model. It rests on a *concretization* function γ, which, following Arquint et al. [15, §4.2], maps each model term to the byte string it stands for. Because different terms (e.g., a guessed MAC and the real one) can have the same bytes, γ has no inverse, and bytes are never mapped back to terms. γ is a homomorphism: each function symbol of the model comes with a function on byte strings, such as `libsignal`'s protobuf encoding for wire messages. The cryptographic symbols use an *interpretation* $\mathcal{I}$, which is universally quantified. The only laws assumed of it are a few trusted facts, chosen so that γ agrees with the equational theory: algebraic and length facts such as Diffie-Hellman commutativity, canonical X25519 public keys, rejection of public keys shifted by a torsion point, and that every AES-CBC ciphertext that decrypts is the encryption of its plaintext. Two further parameters complete γ: byte functions S for *opaque* symbols (SPQR, XEdDSA, Kyber1024 public keys, and decapsulation of non-honest ciphertexts), constrained only to agree with the setup's provisioned bytes, and a per-trace *ghost* g giving the bytes of each fresh value and of the wire messages' `addresses` field, which the model does not represent (Appendix B.3).

A model trace *explains* a hardware trace τ if it is reachable in the model, its events line up with those of τ one to one, in order, with each byte field of a hardware event equal to γ of the corresponding model terms, and it agrees with the execution on details that τ does not record, such as how stored session records replace earlier ones (Appendix B.4).

Theorem	Statement (informal)	Source	Key differences from the source
Forward secrecy	If the attacker knows the key of a message sent or received on the trace, then that key, a chain key it was derived from, or a root key that derived its chain was compromised.	[40, Fig. 3, App. A]	Triple Ratchet modeled as in the paper (SPQR key chosen by the attacker and DR shown no less secure). Whereas the paper's PQXDH composition allows only one session and forbids one-time prekeys, handshake compromises, and ratchet DH-key exposure, we compose with our full PQXDH model: initial root and chain keys are compromised exactly when every handshake deriving them is excused by initiator or responder secrecy, and honest ratchet DH secrets count as compromised when DH is broken. Replay-based FS loss is not ruled out.
Post-compromise security	After a DH ratchet step between two honest DH secrets, a message key derived from it is secret unless a key at or after that step was compromised, or an earlier root key together with one of the two DH secrets.	[40, Fig. 4, App. A]	As for forward secrecy.
Initiator secrecy	Alice's handshake key is secret unless (1–2) Bob's identity key was compromised before the handshake, or at any time together with Bob's signed prekey and either his Kyber prekey or a KEM break; or (3–4) DH was broken before the handshake, or at any time together with Bob's Kyber prekey or a KEM break.	[28, Thm. 7]	PQXDH composed with the Triple Ratchet instead of an abstract AEAD: the paper's single key *SK* becomes `libsignal`'s three HKDF outputs (root, chain, and SPQR keys), so secrecy holds for each output separately unless that specific output was exposed from session state.
Responder secrecy	Bob's handshake key is secret except in the paper's seven compromise cases, which also involve Alice's identity key and Bob's one-time prekey.	[28, Thm. 9]	As for initiator secrecy; also, Bob accepts by verifying the MAC on Alice's initial Triple Ratchet message (carrying an SPQR header and SPQR-salted message keys) rather than an abstract AEAD tag.
Authentication	If Bob accepts a handshake as coming from Alice, then Alice completed an earlier handshake using the same prekeys with some peer (not necessarily Bob), unless Alice's identity key or Bob's signed prekey was compromised, or DH broken, before.	[28, Thm. 8]	Key confirmation via Alice's initial Triple Ratchet message, as for responder secrecy. Kyber prekey agreement comes from binding the public key into the KEM shared secret rather than associated data (PQXDH Revision 2). X25519 prekeys agree up to non-canonical re-encoding (yielding identical DH secrets), since the server can re-encode unsigned one-time prekeys and, if Bob's identity key is compromised, re-sign Bob's signed prekey.
PQXDH Theorem 1 corollaries	Peer authentication; handshake forward secrecy (later identity-key compromises reveal nothing); KCI resistance (compromising Bob's identity key does not let the attacker impersonate Alice to him); session independence (other sessions' leaked prekeys reveal nothing); harvest-now-decrypt-later resistance (a later DH break, without a KEM break or compromises of the session's identity keys or prekeys, reveals nothing).	[28, Thm. 1, App. A]	Stated and proved as five separate theorems rather than informal consequences of Theorems 7–9 (peer authentication and KCI restate Theorem 8). The three secrecy corollaries include the per-key session-state excuse of initiator secrecy.

Table 2 | The ten theorems we prove of the composed model (the PQXDH Theorem 1 row has five). Each holds on every reachable trace. The Source column gives the cited paper's own numbering. Table 5 states them precisely.

Refinement definition. The standard refinement definition would be trace inclusion: every hardware trace is explained by some model trace. That is false here, because hardware bytes are finite. `libsignal` truncates its MAC to 8 bytes, so an attacker who guesses the MAC makes the binary accept a forged message. In the model, the receiver accepts only if the received MAC term is the one it computes, and a guessed literal is a different term, so no model trace explains that hardware trace. Our refinement definition is therefore trace inclusion up to collisions, stated in full as follows.

For every choice W of the abstract primitives inside the hooked routines, every interpretation $\mathcal{I}$ of the cryptographic symbols that meets their contracts and agrees with W, and every well-formed setup r (the parties and their provisioned keys):

$$\exists S.\ \forall \tau \in \mathrm{Traces}_{\mathcal{I},W}(r):\quad \big(\exists g, ms.\ ms \text{ explains } \tau \text{ under } \gamma_{\mathcal{I},S,g}\big)\ \vee\ \mathrm{Collision}_{\mathcal{I},S}(\tau)$$

Here S gives the opaque symbols' bytes; $\mathrm{Traces}_{\mathcal{I},W}(r)$ is the set of hardware traces of the binary from setup r, with the hooked routines computed from W, inversions restricted as in §5.1.3, and the environment restricted as in Appendix B.7; g is a ghost and ms a model trace; and $\gamma_{\mathcal{I},S,g}$ is γ with these three parameters fixed. $\mathrm{Collision}(\tau)$ says that a prefix of τ is explained, and the next run can be explained only through a collision, consisting of two different terms with equal bytes, where at least one term was built by a symbol assumed injective (a fresh value, `dhPk`, `dh`, `kdfExtract`, `symEnc`, `trunc8`, or KEM encapsulation).

Typical collision cases are a received message that the binary accepts only because its MAC equals the expected one, and a plaintext equal to a secret the attacker does not know. Read the other way around, every hardware trace without a collision is explained by a model trace.

The collision case is constrained: S is chosen before the hardware trace, a received message counts as a collision only if it collides under every explanation of the prefix that has not already read a colliding message, and never if a term the attacker knows explains it without itself colliding. It also excuses a Kyber decapsulation that disagrees with encapsulation, since no Kyber correctness law is assumed on $\mathcal{I}$. This clause depends only on the hooked Kyber routines, Bob's provisioned Kyber key and the received ciphertext, so it holds only if decapsulation under that key disagrees with encapsulation: the Kyber implementation is incorrect, the provisioned key is malformed (key generation is not verified), or the ciphertext is one of the decryption failures that Kyber1024 has with negligible probability (Appendix B.5). Bounding the probability of Collision is a computational question, outside Lean. Arquint et al. instead assume collision freedom as a hypothesis [15, §4.3].

5.1.5. Main Theorem

Our main symbolic result is that every hardware trace of `libsignal-cli` that does not reach a collision is explained by some model trace, and that all ten theorems of Table 2 hold on *every* model trace that explains it, against an attacker that knows only what it observed in the hardware trace.

To state it, let $Q(tr, K)$ be the conjunction of the queries of the theorems of Table 2 on model trace tr with attacker knowledge K. The model-level theorems together state $\mathrm{Sec}(Model)$: $Q(tr, \mathrm{Knows}(tr))$ for every reachable model trace tr, where Knows is the attacker knowledge of §5.1.1. The binary-level property $\mathrm{Sec}(Asm)$, where Asm is the machine code of `libsignal-cli`, is the statement of the following theorem.

Theorem 1 (Binary symbolic security)**.** *For every choice W of the abstract primitives inside the hooked routines, every interpretation $\mathcal{I}$ of the cryptographic symbols that meets their contracts and agrees with W, and every well-formed setup r (two honest parties, initiator Alice and responder Bob, their provisioned*

keys, and each party's pinned copy of the other's identity key),

$$\exists S.\ \forall \tau \in \mathrm{Traces}_{I,W}(r) \textit{ with fresh coins}:$$
$$\big(\mathrm{Expl}(\tau) \neq \emptyset \ \wedge\ \mathrm{Safe}(\tau)\big) \ \vee\ \exists \tau_1, e, \tau_2.\ \tau = \tau_1\, e\, \tau_2 \ \wedge\ \mathrm{Collision}_{I,S}(\tau_1, e) \ \wedge\ \mathrm{Safe}(\tau_1),$$

where $\mathrm{Safe}(\tau) := \forall (g, ms) \in \mathrm{Expl}(\tau).\ Q(ms, K_{ms})$.

$\mathrm{Expl}(\tau)$ is the set of pairs (g, ms) such that ms explains τ under $\gamma_{I,S,g}$; requiring it to be nonempty stops the $\forall$ from holding vacuously. $\mathrm{Collision}_{I,S}(\tau_1, e)$ says that τ_1 is explained and that the next run e can be explained only through a collision, as in §5.1.4. A trace has *fresh coins* if the setup's X25519 secrets and the secrets that its successful runs draw from their `coins` (X25519 keys and Kyber encapsulation randomness) are pairwise distinct, compared after X25519 clamping. The theorem also establishes progress: every reachable valid call exits or aborts, and never faults or diverges. The theorem differs from Sec(*Model*) in four ways. First, a trace that reaches a collision gets the guarantee only on its explained prefix before the colliding run. Second, the knowledge K_{ms} is the same Dolev–Yao closure, but seeded only with the setup's public values and the terms matched with what the attacker observed: the bundles, the wire messages, and the leaked or inverted secrets. Third, the hardware traces fix a narrower scenario than the model (§5.1.6). Fourth, the theorem assumes fresh coins; if the coins come from a CSPRNG, this fails with probability about $k^2/2^{251}$ over k draws.

Proof sketch. The proof is a mostly a standard refinement-preservation argument:

$$\mathrm{Sec}(\textit{Model}) \ \wedge\ \textit{Asm} \sqsubseteq \textit{Model} \implies \mathrm{Sec}(\textit{Asm})$$

where $\textit{Asm} \sqsubseteq \textit{Model}$ is the refinement definition of §5.1.4. Every model trace that explains a hardware trace is reachable, so the model-level theorems hold on it, and the hardware attacker knows no more than the model attacker on that trace. Since each query uses the attacker's knowledge only as a hypothesis, an attacker that knows less cannot break it. In the collision case, the same argument gives the queries on every explanation of the prefix before the colliding run (Appendix B.8).

5.1.6. Limitations

Cryptography. Our guarantees are symbolic: they assume perfect cryptography and say nothing about the computational secrecy of bytes. At the binary level, the theorem excuses byte collisions but does not bound their probability, and it trusts the hooked routines of §4 to meet their contracts and the low-level cryptographic ones to compute an interpretation that satisfies a few algebraic laws. It also assumes fresh coins (§5.1.5). Since no Kyber correctness law is assumed, a run where decapsulation under Bob's provisioned key disagrees with encapsulation falls under the collision clause and gets no guarantee; this requires an incorrect Kyber implementation, a malformed provisioned key, or a decryption failure that occurs for Kyber1024 with negligible probability (§5.1.4, Appendix B.5).

Scenario. The assembly-level theorem covers a narrower scenario than the model (Appendix B.7 lists its main restrictions). XEdDSA and SPQR are opaque in γ, so we assume an authentic PKI and SPQR fields copied from honest messages: both parties start with each other's identity key pinned, so `libsignal`'s trust-on-first-use path is not exercised, and the bundle Alice receives carries Bob's genuine identity key, signed prekey and Kyber prekey. The harness provisions two honest parties with fixed roles and one session per session file, and gives Bob one signed prekey, one reusable last-resort Kyber prekey and at most one one-time prekey; one-time Kyber prekeys are not covered. It also bounds input sizes and the number of sends, requires `coins` files of exactly 192, 64 and 32 bytes for `init`, `send` and `recv`, and restricts Bob after a non-matching prekey message. A run that aborts (for instance on a failed allocation) ends the trace.

Attacker. Leaks reveal only long-term and prekey secrets and the keys of a party's current session; side channels and other secrets in memory are out of scope. An exact replay of a prekey message counts as the same session in the model, so our forward secrecy and post-compromise security theorems do not rule out the replay-based loss of forward secrecy described by Cheval et al. [40, §3.2.2–3.2.3]: the binary runs `libsignal`'s deduplication of base keys, but the model does not require it.

Relation to prior analyses. We replicate only the symbolic results of the papers: the computational theorems of Bhargavan et al. have no counterpart here (see §5.2.3 for our computational guarantees), and we do not model header encryption, which `libsignal` does not use. Finally, as in Cheval et al., our ratchet theorems let the attacker choose the SPQR keys, so they show that the Triple Ratchet is no weaker than the Double Ratchet, not the post-quantum healing that SPQR adds. In particular, once DH is broken, at any point in the trace, the ratchet's DH secrets count as compromised, so post-compromise security claims no healing against an attacker who can break DH, even one who cannot break the KEM.

5.2. Computational Proofs

We state computational security as a game [88, 22] between a *challenger* and an *adversary*. The challenger samples the secrets and holds every party's state, and the adversary interacts with it through *oracles* that run protocol steps, corrupt parties and pose challenges. The adversary's *advantage* measures how well it does in violating some security property, for example, its probability of getting a forgery accepted, or its probability of guessing a hidden random bit (minus the trivial 1/2 probability of blindly guessing). If the probability of guessing a hidden bit is, say, $1/2 + 2^{-64}$, we conclude that the bit is effectively secret.

Security is proved via *reductions*. A reduction R to an assumption, such as the hardness of Gap Diffie-Hellman (Gap-DH), is an algorithm that uses the adversary A as a subroutine to violate the assumption. R runs A, answers A's oracle queries itself so that A appears to be attacking the protocol, and turns A's advantage into a solution of its own challenge (say, a challenge instance of Gap-DH). A proof shows that the advantage of A in attacking the protocol is bounded by some function of the corresponding advantage of various reductions R in violating different assumptions, and statistical terms (such as the probability of a collision). If every hardness assumption holds then all reductions have small advantage and we infer that A has small advantage in breaking the protocol. Importantly, the reductions must be efficient for the claimed bounds to be meaningful.

Proofs usually proceed through a sequence of *game hops*, each a small change to the game justified by one reduction or one statistical argument. What is new in our setting is that the oracles of our games run the compiled `libsignal-cli` executable.

The proof connects three layers:

1. **A security game** for PQXDH composed with the Triple Ratchet. The adversary runs many handshakes, turns some of them into ratchet sessions, and attacks the confidentiality and authenticity of ratchet messages. It may corrupt long-term keys and ratchet state, and interleave all of these queries arbitrarily.
2. **A byte-level model** of `libsignal` in Lean. The model follows the Rust source closely, down to encodings, key-derivation labels, key schedules and the SPQR state machine.
3. **The AArch64 machine code** of `libsignal-cli` (§4). Every ratchet step in the game is one run of this executable in the directory of the party concerned, from its entry point to its final `exit_group` system call.

We prove that the machine code computes exactly what the model computes, and that the game run

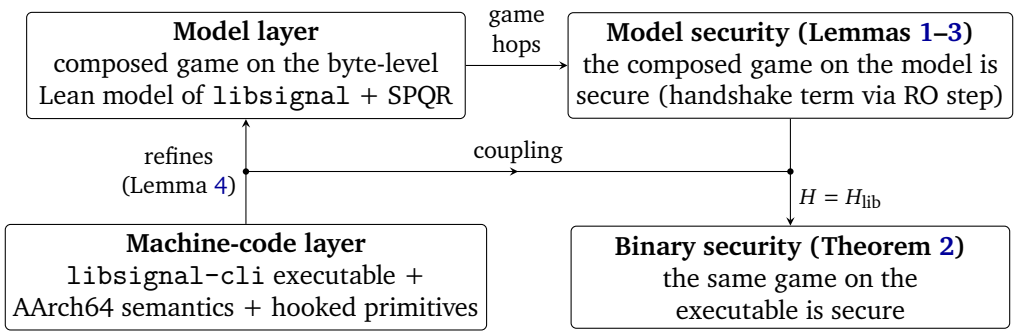


Figure 7 | Overview of our computational proof. Lemma 4 establishes that every run of the executable has the outcome the model predicts. A coupling lifts it to the game. Lemmas 2–3 bound the model game by the key indistinguishability of its handshake and single-session security. Lemma 1 bounds the key indistinguishability of the one-key random-oracle handshake H_{ro}; one random-oracle idealization step relates it to `libsignal`'s handshake H_{lib} (§5.2.3).

on the model is secure. A coupling argument then shows that the game run on the machine code is secure as well (Figure 7).

5.2.1. The Security Game

Our game combines two games from the literature. The *handshake oracles* are those of the PQXDH key-indistinguishability game of Fiedler and Günther [52]. They let the adversary register honest parties and parties of its own, generate prekeys, initiate and accept sessions, corrupt identity keys and reveal session keys. The *ratchet oracles* are those of the secure-messaging game of Alwen, Coretti and Dodis [7] (ACD19). They let the adversary send and deliver messages, pose a challenge (encrypt one of two messages of its choice), inject forged messages and corrupt either party's ratchet state.

A single oracle connects the two halves. The adversary may *promote* a session that it initiated through the game into a ratchet session. The oracle first checks that the handshake used keys that an honest party published, and that no earlier session used the same ephemeral key. It then starts the Triple Ratchet, including SPQR, for both parties with the *real* session key of the handshake. Finally, it marks the responder's signed prekey as revealed in the handshake game: the responder's first ratchet key is that prekey, so the ratchet needs its secret. The ratchet proof therefore does not rely on that secret staying hidden.

The adversary wins if it guesses which of its two challenge messages was encrypted, or if it gets a forged message accepted. As in ACD19, the game rules out trivial wins, such as decrypting a challenge with state the adversary has just stolen. The game is written once, for any implementation of the handshake and ratchet steps. We write $\mathrm{Adv}^{\mathrm{comp}}(A)$ for the advantage of adversary A in the game run on our Lean model, and $\mathrm{Adv}^{\mathrm{bin}}(A)$ for its advantage in the game run on the executable. Our bounds are concrete: they are stated in terms of *budgets* that limit the numbers of parties, prekeys, sessions and attacker's queries (Theorem 2). Appendix C.1 defines the game precisely.

5.2.2. Reductions as Programs

Our theorems assert that efficient reductions exist, so we must make precise what counts as a reduction. One could express reductions as general-purpose Lean functions. However, this choice makes statements of the form "there exists a reduction" vacuous: a Lean function can be noncomputable, for instance by picking the secret key by choice, or arbitrarily expensive.

We therefore write reductions as programs in a small, deeply embedded *reduction language* with a cost semantics. A reduction runs the adversary exactly once, as a black box and without rewinding, and answers each of its oracle queries by running a *handler*. If the assumption game has a challenge phase, the reduction may pause the adversary at one query, obtain the challenge, and then resume the same run.

A handler computes on ordinary data, samples coins and queries the oracles of the assumption game. It can operate on keys and group elements only through declared primitive operations, and it loops only up to bounds written in the program. The cost of a reduction is computed from its program text alone.

A reduction is *cost-bounded* if its cost, its number of oracle queries and the sizes of the values it computes are bounded by explicit polynomials in the game's budgets, independent of the adversary, and if its program fits a fixed size budget. This rules out a "reduction" that breaks the assumption on its own, for example by exhaustive search or by hard-coding a large table such as a table of discrete logarithms. Appendix C.2 gives the cost semantics.

5.2.3. Main Theorem

We consider two versions of the PQXDH handshake, both with Kyber1024 as the KEM. The first, H_{lib}, is `libsignal`'s handshake. Like `libsignal`, it uses a single Curve25519 scalar per party for both X25519 DH and XEdDSA signatures [79], and it derives the session key with HKDF [68], the standard hash-based key-derivation function. The second, H_{ro}, is the same handshake with that HKDF replaced by a *random oracle* [21]: an ideal, truly random function that every party, including the adversary, can query. (Prior analyses [52, 28] also model this HKDF as a random oracle.) Our game runs H_{lib}.

Our main theorem states that the game run on the executable is secure. Concretely, it gives a bound on $\mathsf{Adv}^{\mathrm{bin}}(A)$, the advantage of any adversary A that respects the budgets. The bound is a sum of two kinds of terms. The first kind is the advantage of a reduction, built from A, against one of our cryptographic assumptions. The second kind is a statistical term, such as the probability of a collision or of a decryption failure. One term, $\varepsilon_{\mathrm{mlkem}}(A)$, depends on the adversary itself (see below). The reductions are fixed by the theorem and are cost-bounded, so if every assumption holds, every term is small and no efficient adversary breaks the executable. Clause 1 states this bound for `libsignal`'s handshake as is. Clause 2 tightens the handshake term when the handshake's HKDF is modeled as a random oracle.

Theorem 2 (Binary security, simplified)**.** *Let N_{id} be the number of honest parties, N_{spk} and N_{opk} the numbers of signed and one-time prekeys, N_{send} the number of initiated sessions (of which $N_{\mathrm{send,opk}}$ use a one-time prekey), q_{on} the number of handshake queries other than random-oracle queries, q_{start} the number of promoted sessions, and q_k the number of ratchet queries in the k-th session.*

1. Security. *There are cost-bounded reductions such that, for every adversary A that respects these budgets, and under functional premises on the primitives (Appendices C.7 and C.8),*

$$\mathsf{Adv}^{\mathrm{bin}}(A) \;\le\; 2\,\varepsilon_{\mathsf{KI}} \;+\; \sum_{k<q_{\mathrm{start}}} \mathsf{Sess}_k \;+\; 2\,N_{\mathrm{send}}N_{\mathrm{spk}}\,\delta_{\mathrm{kem}} \;+\; 2\,\varepsilon_{\mathrm{mlkem}}(A),$$

   where δ_{kem} is the probability that Kyber1024 decapsulation fails, $\varepsilon_{\mathsf{KI}}$ is the advantage of a reduction against the key indistinguishability of the handshake, Sess_k bounds the advantage of the adversary in the k-th ratchet session, and $\varepsilon_{\mathrm{mlkem}}(A)$ is the probability that an ML-KEM-768 decryption failure occurs in the game.
2. *If HKDF is modeled as a random oracle in the handshake, then the bound holds with $\varepsilon_{\mathsf{KI}}$ replaced by $\beta^{\delta}_{\mathsf{KI}} + \varepsilon_{\mathrm{ro}}$ (defined below).*

The terms of the bound are as follows.

- $\varepsilon_{\mathsf{KI}}$ is the key-indistinguishability advantage, against H_{lib}, of a reduction run with A. Under the random-oracle idealization of clause 2, $\varepsilon_{\mathrm{ro}}$ is the gap between H_{lib} and H_{ro}, and $\beta^{\delta}_{\mathsf{KI}} = \beta_{\mathsf{KI}} + 2\,N_{\mathrm{send}}N_{\mathrm{spk}}\,\delta_{\mathrm{kem}}$ bounds the key indistinguishability of H_{ro}. Here β_{KI} is a sum of the advantages $\varepsilon_{\mathrm{Sig}}$, $\varepsilon_{\mathrm{ShCr}}$ and $\varepsilon_{\mathrm{GapDH}}$ of cost-bounded reductions against unforgeability of XEdDSA, collision resistance of Kyber1024 (SH-CR [28]) and Gap-DH on X25519, plus public-key collision terms. Each advantage is multiplied by the number of parties, prekeys or sessions that the adversary's online handshake queries can touch, or by products of these numbers, so the term grows with the adversary's activity rather than with the size of the deployment (Lemma 1 gives β_{KI} in full).
- Sess_k bounds the advantage of the adversary in the k-th ratchet session as if that key were uniform; the term $2\varepsilon_{\mathsf{KI}}$ pays for replacing the real handshake keys by uniform ones. Following ACD19 [7], it sums the advantages of cost-bounded reductions against the continuous key agreement (CKA) of the DH ratchet, the root key derivation as a PRF-PRNG, and the forward-secure encryption of messages, with further terms for SPQR's use of HKDF. The remaining terms are statistical. The largest is the probability of forging `libsignal`'s 8-byte MAC tag, at least 2^{-64} per attempt, which grows quadratically in both the number of messages and the number of epochs of the session. Appendix C.9 gives Sess_k in full.
- For δ_{kem}, the Kyber specification gives the heuristic estimate $\delta_{\mathrm{kem}} \le 2^{-174}$.
- $\varepsilon_{\mathrm{mlkem}}(A)$ is the probability that, in the game run with A, the coins of one or two sends, honest or chosen by A, give an ML-KEM-768 key pair and encapsulation whose decapsulation fails. Since A chooses the coins of the sends it injects, this term depends on A: it is small unless A can find ML-KEM-768 decryption failures. It is not the advantage of a reduction, because a reduction fixed in advance could hard-code a failing pair.

Proof sketch. The proof has four main parts, stated as lemmas in Appendix C.3.

1. *PQXDH* (Lemma 1) bounds the key indistinguishability of H_{ro} by $\beta^{\delta}_{\mathsf{KI}}$. Its proof is a sequence of game hops in the style of [52].
2. *Single session* (Lemma 2) bounds the advantage of an adversary against one ratchet session, started from a uniform key, by Sess.
3. *Composition* (Lemma 3) reduces the game run on the model to the key indistinguishability of the handshake and to one single-session game per promoted session, by a hybrid argument over the sessions.
4. *Refinement* (Lemma 4) states that every run of the executable has the model's outcome, unless the machine runs out of memory. Its proof is compositional, in a separation logic for AArch64. Each compiled function, from musl's start-up code and its `mallocng` allocator through the CLI, protobuf, `libsignal` and SPQR, gets a contract that relates its registers and memory to the corresponding model function. The contracts are proved by symbolic execution. The lemma holds for every placement of the code in memory that matches the executable, so addresses and relocations are checked against its bytes rather than trusted from the linker.

The theorem follows from the four lemmas by coupling the two games. The game on the executable stores directories where the game on the model stores model states, and the coupling maintains that each directory is the encoding of the corresponding state. By Lemma 4, together with a proof that no run in the game exhausts memory (§4.1), this invariant breaks only if a Kyber1024 decapsulation fails in an honest handshake, which gives the δ_{kem} term. Otherwise, the adversary wins both games with the same probability. ML-KEM-768 decryption failures in SPQR are charged once for the whole game: the Lean proof carries out the composition in a game that stops at the first such failure and bounds the probability of reaching it by $2\,\varepsilon_{\mathrm{mlkem}}(A)$.

5.2.4. Assumptions and Idealizations

Cryptographic assumptions. All but two of our assumptions are standard. The two exceptions arise because `libsignal` uses each party's long-term Curve25519 secret key for two purposes: to compute X25519 Diffie-Hellman shared secrets in the handshake, and to sign the party's prekeys with XEdDSA. Standard assumptions treat each primitive on its own, with its own key, and say nothing about whether using one key for both could help an attacker. For example, signatures on chosen messages might help it compute DH secrets, or DH results might help it forge signatures. We therefore use *joint-security* assumptions [46, 95]: XEdDSA signatures remain unforgeable even for an attacker that can also obtain DH results for the same key, and Gap-DH remains hard even for an attacker that can also obtain signatures under it.

Outside the trusted base, we prove in Lean that both joint assumptions reduce to standard Gap-DH when SHA-512 is modeled as a random oracle (Appendix C.10). The main theorem keeps the joint assumptions, because deriving them from Gap-DH would weaken the bound considerably. Finally, ML-KEM-768 decryption failures are not covered by an assumption; the bound charges them directly as $\varepsilon_{\text{mlkem}}(A)$ (§5.2.3).

Random oracle. Clause 2 of the theorem treats the handshake's HKDF as a random oracle. This is a heuristic: real HKDF is a fixed function, so clause 2 is not a theorem about it. Clause 1 makes no such idealization, and none of the ratchet's key derivations are idealized in either clause.

Environment. The routines that `libsignal-cli` calls through contracts (§4) are assumed to behave as their Lean specifications say. The theorem also assumes functional properties of the primitives, such as that decryption inverts encryption, and fixes features of the process environment, such as the clock, the kernel's random bytes, the stack size and the available memory (§4.1). Appendix C.5 lists these assumptions. Running out of memory is not assumed away: a proof shows that no run in the game exhausts the available memory (§4.1).

5.2.5. Limitations

Our bounds give no guarantee against a quantum adversary. The symbolic proof covers part of this gap (§5.1). The game also restricts the adversary in ways that the literature does not. The most significant restriction is that the adversary cannot inject its own PQXDH initial message to the responder of a session that is later promoted. Further, the theorem does not cover the privacy of the payload, or the authentication of the responder, for first messages of handshakes that are never promoted. As in ACD19, corruption is refused at some points, for instance while an uncompromised message is in transit. Our game also refuses it until the initiator reaches epoch 10, since until then she may still hold the receiving chain for the leaked signed prekey.

The setting is simplified as well. Each promotion starts one session, messages have a fixed length, and there is no multi-device state. Identity keys are trusted on first use. The statistical terms limit the bound to moderate scale: if nearly every message starts a new epoch, they make it vacuous for single sessions of about 25,000 messages, or for 2^{20} sessions of 2^{10} messages.

We note that tightening bounds like this is a natural task for AI. Because the theorem's statement is a Lean definition, a tighter bound is a change to the specification, which humans review, followed by a new proof campaign, which agents carry out.

Finally, the results concern our build of `libsignal-cli`, run as a fresh process on a machine with the memory of §4.1, not Signal's own applications, and some of the environment assumptions

are argued informally. Appendix C.6 lists the remaining restrictions.

5.3. Trusted Computing Base

The trusted computing base (TCB) is the code that must be audited in order to have confidence that the theorems provide the security guarantees that we claim. To believe our theorems, a reader must check their *statements* and, transitively, every definition they use. For the symbolic proof, this is the DyLean model and its security statements, the event layer, the concretization function γ, the refinement statement with its collision clause, and the environment's restrictions (§5.1.6). The computational proof trusts the definitions of the game, the assumption games and the bound, the Lean model of `libsignal` and SPQR, and the random-oracle step (§5.2.4).

Both proofs trust (a) the AArch64 semantics and the Lean model of how Linux runs a process (§2.2, §4.1), (b) the contracts of the hooked routines (§4), and (c) the functional and algebraic properties assumed of the cryptographic primitives. These are definitions that are transitively included in the theorem statements. The TCB also includes the tooling that extracts symbol names from the compiler's object files, and the scripts that build the binary in a reproducible way (Appendix C.6).

Everything else is checked by Lean, including the model-level proofs, the refinement, preservation and coupling arguments, the game hops and reductions, the function contracts and the program logic. For this reason, trusting our proofs also requires trusting Lean itself. The Lean community has invested significantly in trustworthiness of the Lean system, including the design of a small, carefully audited kernel, as well as independent kernels and consistency proofs [36, 37, 16]. Notably, however, the Rust compiler is *not* trusted. An error in any of the untrusted components that compromises the security properties that we formalize, including a bug in the compiler, or a bug in the Rust code, would cause the theorems to become false, so that the proof would fail.

6. Finding Historical Bugs

To check whether our specifications are strong enough to catch real bugs, we reintroduced two bugs that Cheval et al. [40] found in `libsignal` and that are now fixed. We injected each bug by reverse-applying its upstream fix commit to the `libsignal` v0.105.4 source and recompiling the binary, which we call Asm_{bug}.

- *Weak state* [40, §3.2.1], fix `cd36118`: a prekey message that fails to decrypt on its new session is trial-decrypted with older sessions, and if that succeeds the new session's unconfirmed state, which holds a copy of the signed prekey secret, is stored.
- *Small subgroup* [40, §3.2.2], fix `5293caa`: the PQXDH responder does not check that the initiator's X25519 base key is torsion-free and in range, so a replayed handshake with a mangled base key creates a second session with the same shared secret.

For both bugs we have complete Lean proofs that the symbolic security theorem (§5.1.4) does not hold of the bug-injected binary, $\neg\mathrm{Sec}(Asm_{\text{bug}})$. Recall that $\mathrm{Sec}(Asm)$ requires that, for every extern interpretation $\mathcal{I}$ meeting the contracts and every well-formed setup r, there exist opaque-symbol byte functions S such that every collision-free hardware trace $\tau \in \mathrm{Traces}_{\mathcal{I},W}(r)$ is explained by at least one reachable model trace ($\mathrm{Expl}(\tau) \neq \emptyset$) and satisfies all ten security queries of Table 2 on every such explanation. Each disproof exhibits concrete witness inputs and splits into two parts: a specification part showing that, under a valid extern interpretation $\mathcal{I}$ and well-formed setup r, no choice of S either explains the resulting hardware trace τ, which has fresh coins, in the model or excuses it as a collision; and an implementation part showing that Asm_{bug} produces τ under the AArch64 semantics (§2.2).

For the small-subgroup bug, Alice initiates a session and sends a prekey message whose X25519 base key the attacker modifies by setting bit 255, giving a distinct encoding with the same Diffie-Hellman outputs; the model's canonical-key check rejects this message, whereas Asm_{bug} accepts it and stores a session. For the weak-state bug, Alice establishes a session with Bob (leaving skipped message keys in Bob's session file) and then initiates a second session; the attacker splices the second session's PQXDH header onto an inner message from the first session, which Asm_{bug} accepts via trial decryption against the stored session while no model transition explains the run. Both disproofs need no assumptions beyond Lean's standard axioms: the Lean kernel parses the SHA-256-pinned ELF binary Asm_{bug} and steps its AArch64 machine code across the runs of each witness trace.

On the other hand, refuting the computational theorem is much harder. The theorem states that there exist cost-bounded reductions, against hardness assumptions such as Gap Diffie-Hellman, whose success probabilities bound the advantage of every adversary. To refute it, one must exhibit an adversary that exploits the bug for which *every* cost-bounded reduction rarely succeeds, which amounts to proving the hardness assumptions themselves, an open problem.

7. AlphaProof Code in Practice

We now describe how we used AlphaProof Code in our verification of `libsignal`. The proof work was done by a team of five over four weeks, working in parallel on symbolic and computational proofs. Each of us ran agents nonstop on our own workstation, directing specification work and launching proof campaigns on different parts of the system.

Scale. The repository contains nearly 7 million lines of Lean, counting comments and blank lines. About 6.5 million of them are proofs: 5.3 million about the binary, which both security proofs share, about 860,000 for the computational proof and about 350,000 for the symbolic proof. Another 140,000 lines prove the reintroduced bugs (§6), and 70,000 verify the allocator. The verified `libsignal` code is about 560 KB, about 140,000 AArch64 instructions. The specification directories hold about 54,000 lines, but only part of them is trusted: every file is marked as trusted, trusted in part, or checked by Lean. Reading every definition that our headline theorems depend on means reading about 18,000 lines of code in the repository (about 28,000 with documentation), plus about 8,000 lines of the AArch64 semantics and a Lean representation of the compiled machine code (about 95,000 lines), which a tool extracts from the compiler's object files; Lean checks its code against the executable's bytes, so only the symbol names it assigns need to be trusted.

Writing specifications. The most important lesson from our experience is that, once proofs become cheap, the hard question is *what* to prove. The majority of our effort went not into proofs but into specifications. This trend is perhaps apparent in software development more broadly, but it is particularly acute for cryptographic protocols, where an entire literature is devoted to the question of how to formalize security (§8). One possible future is that programmers write formal security specifications, natural-language intent and performance constraints, and AI generates, optimizes and proves the implementation.

Our specifications were developed through an intense back-and-forth between humans and agents powered by Gemini 4 Carbon and Claude Opus 5.5. For each component—for example, the binary contracts of a `libsignal` entry point, or the computational main theorem—the specification writer drafted a small set of top-level statements, following a model from the literature where one existed. An agent from a different model family then reviewed the draft for faithfulness to the source paper and to `libsignal`, for strength (would it rule out known or plausible bugs?), and, where it goes

beyond the literature, for consistency with the literature's conventions. We gave many rounds of targeted feedback before approving a specification and pinning it as the target for proofs. Left to themselves, agents produced specifications that were hard to audit; we sometimes directed rewrites for legibility, which paid off when new team members joined late to review them.

Writing specifications against real code also turned out to be a stress test for the literature. For both formalizations we record every place where the Lean statements depart from their reference papers or from `libsignal`, together with its effect on the result; the computational catalog has over 200 entries and the symbolic one about 35. Most are routine: the same idea stated differently in Lean, a place where `libsignal` differs from the scheme analyzed on paper, or a restriction the papers needed for their tools to terminate, which our symbolic model lifts. Some, however, are errors in the published analyses that we had to repair before the proofs could go through. Fifteen are gaps or bugs in the computational analysis of the Double Ratchet [7], such as missing hypotheses and security game conditions that allow trivial wins. On the symbolic side, the printed ProVerif queries of Cheval, Jacomme and Richards [40] look up skipped message keys at the wrong ratchet index, which makes receiver-side forward secrecy false as printed, leave a variable unbound, and bracket post-compromise security so that exposing a party's own DH key, together with any root-key compromise, excuses a message regardless of when the heal happened. In the PQXDH analysis of Bhargavan et al. [28], reading "some SPK_B was compromised" literally makes the session-independence corollary unprovable; we bind it to the session's own signed prekey, as their ProVerif query does.

Generating proofs. The proofs were produced entirely by agent teams powered by Gemini 4 Carbon (§2.3), and none of the authors wrote any proof code. Keeping dozens of agents productive for days at a time is an engineering problem in its own right, and much of our early effort went into the skills that define the agent team. We developed them iteratively by watching campaigns run. A separate agent, above the team orchestrator, serves only as the interface between us and the team: it monitors the team, and we used its logs together with it to diagnose bottlenecks and propose fixes to its processes. Early on, monitoring the team took a lot of human time; this dropped as the skills matured. In the near future, we expect teams like this one to improve themselves.

We summarize a few improvements that emerged through human-AI iterative co-evolution in this way. Agents share notices through a common bulletin file rather than repeating messages. Team leads try to refute their goals before assigning work, fix shared parameters such as stack bounds across callees, and point workers to existing lemmas. Support agents batch expensive Lean builds to fit in memory, watch for stalled agents and exhausted model quotas, and refactor slow modules. The orchestrator reshapes the team as the campaign evolves. Every new invariant an agent introduces must come with a witness that it holds on some reachable state, so that agents cannot make progress by proving vacuous lemmas.

While no human wrote proof code, compute latency was an issue. A clean build of our proofs could take several hours, and early in the project, small source changes or a recompiled binary required large proof repairs. Better program-semantics infrastructure later reduced this. We see substantial room for optimization of proofs generated by and proof techniques deployed by AI.

Preventing specification weakening. Agents under pressure to finish a goal will sometimes weaken it [107, 78], for example by adding a hypothesis or narrowing a precondition, and this can happen indirectly through a definition that the statement depends on: in one swarm of Lean-proving agents, an agent redefined a predicate used in a conjecture's hypothesis as `False`, and the trick spread to the rest of the swarm [78]. We therefore never rely on an agent's own report that a goal is done. The gatekeeper (§2.4) sees only file paths and theorem names, and none of the provers' dialogue or

explanations; Lean replays every proof in a fresh environment, and `sorry` and `native_decide` are rejected.

At the same time, freezing the first draft of a specification is too rigid. Even experienced specification writers rarely get every detail right on the first try, and first drafts are often overly conservative in benign ways that make them false of the real code. Our solution is the amendment process of §2.3. An amendment is allowed only if the proof cannot instead be completed with a stronger invariant or a different decomposition, and only if it keeps the non-vacuity witnesses and honest runs covered. A neutral amendment must bring the specification closer to the code. A proposal that would weaken security to match the code is never accepted as an amendment; it is raised to a human as a potential bug. Accepted amendments are small and logged with their justification on an audited branch. Amendments that change trusted assumptions are approved during the campaign, by one of us or under a policy we set (§2.3), and all amendments are reviewed by us at the end of each campaign. One example is extending the composed symbolic query's handshake-compromise exception from the initial root key to the initial chain key derived alongside it.

The human role. To summarize, we decided the scope and order of the work, chose the reference models, and guided and approved the trusted statements. We also designed the structure of the specifications and guided the overall proof strategy: for example, writing reductions in a small reduction language with a cost semantics (§5.2.2), and, for the symbolic proof, an event layer and concretization homomorphism that maps binary executions to protocol events, together with a refinement to a DyLean model up to collisions that preserves the security properties (§5.1.4). During proof campaigns, we reviewed escalated specification issues and potential bugs, adjudicated amendments that change trusted assumptions, and monitored progress through status logs. We also designed the skills that define the agent team, and refined them as we observed the agents' behavior. We did not write proofs. Every statement we present as a theorem is checked by Lean, and every assumption we rely on is listed among the trusted definitions (§5.3).

8. Related Work

Analyses of the Signal protocol design. Kobeissi, Bhargavan and Blanchet [67] analyzed a variant of Signal with ProVerif and CryptoVerif. Bhargavan et al. [28] verified PQXDH with ProVerif and CryptoVerif, finding issues in an early version of its specification. Cheval, Jacomme and Richards [40] gave an automated analysis of the Double Ratchet composed with PQXDH, found the two `libsignal` bugs we re-inject in §6, and proved forward secrecy and post-compromise security. Our symbolic theorems restate theirs and those of Bhargavan et al. with adaptation to connect to our `libsignal-cli` binary.

On the computational side, Cohn-Gordon et al. [42, 43] gave the first game-based analysis of Signal's X3DH handshake and Double Ratchet as a multi-stage key exchange. Alwen, Coretti and Dodis [7] defined secure messaging with immediate decryption and proved the Double Ratchet secure in a modular way. Our ratchet game follows their definitions.

Fiedler and Günther [52] analyzed PQXDH in a key-indistinguishability game, which our handshake oracles follow. Both Fiedler and Günther [52] and Bhargavan et al. [28] give each party separate Diffie-Hellman and signing identity keys; Lemma 1 bounds the one-key handshake of `libsignal` under joint X25519 Gap-DH and XEdDSA unforgeability assumptions [46, 95, 35], while standalone Lean proofs for the two-key key-indistinguishability and first-message authentication models remain in the artifact outside the trusted base.

Verified protocol implementations. Several projects verify implementations rather than designs. miTLS and Project Everest [27] implemented TLS in F# and proved computational security of the implementation, Noise* [59] generates verified implementations of Noise protocols, and LibSignal* [83] is a verified implementation of the Signal protocol written in F* and compiled to WebAssembly. These are new implementations written for verification; LibSignal*, in particular, is separate from the production `libsignal`.

Deployed code has been verified for narrower properties: the s2n TLS library is continuously verified with SAW [41] for functional correctness of its HMAC, DRBG and handshake state machine, and SPQR's production Rust crate is proved panic-free with hax [25, 90], alongside ProVerif models of its design [89]. Neither proves protocol-level security of the code. One line of work extracts a model from implementation code and verifies the model: fs2pv [26] translates F# code into ProVerif models, Aizatulin et al. [2] extract ProVerif models from C code by symbolic execution, ASPIER [38] abstracts C code into a model that it checks for a bounded number of sessions, and Kobeissi et al. [67] verified a JavaScript implementation of a Signal variant by translating it to ProVerif and CryptoVerif models.

Notaby, another line of work annotates the source code itself with specifications and proves them with a type checker or program verifier. Arquint et al. [15] verify annotated Go and Java implementations against a Tamarin model, with a soundness argument that maps model terms to bytes; we follow their concretization in our symbolic refinement. Their main case study gives an end-to-end symbolically verified implementation of the WireGuard key exchange, derived from the official Go implementation of this widely deployed VPN and interoperable with it. To make verification tractable, they modified that code: they removed DDoS protection, metrics and load balancing, and adapted it around stubs for cryptographic and network operations. We verify the `libsignal` code unmodified. Dupressoir et al. [50] prove protocol properties of annotated C code with a general-purpose verifier.

All of these work either on the source level, on code written or modified for verification, on code in a verification-friendly language, or with a model extracted from source, so the guarantee for the running program relies on a trusted compiler, extraction step or language semantics. We verify the unmodified production library as it is compiled by the stock Rust compiler in a CLI harness (with hooked cryptographic primitives), and state our theorems over machine code. This removes the compiler and the Rust semantics from the trusted base, at the price of reasoning about optimized AArch64.

Mechanized symbolic model cryptography. Symbolic analyses make idealized assumptions about the underlying cryptographic primitives and model the attacker as a Dolev–Yao network adversary that manipulates messages as terms [49]. This focuses the analysis on the protocol design rather than the cryptographic algorithms employed by the protocol. In ProVerif [30, 32], protocols are specified in an applied pi-calculus (process calculus), and proofs are done by translation into Horn clauses, and attacker knowledge is represented as a predicate over these clauses. It permits reasoning about an unbounded number of sessions, but false positives are possible. Tamarin [76, 86] models state as multisets, protocols as multiset rewrite rules, and attacker knowledge as one kind of fact over multisets. Properties are stated in first-order logic over traces. Proofs are solved by constraint solving either automatically or auto-actively. DY* [24] proves trace invariants of executable protocol code in F*, and two recent Lean frameworks build on its approach: DyLean [99], which adds user-defined equational theories and threat models, and LeanDY [63], which combines type-based and trace-based reasoning. Our symbolic specifications and proofs use DyLean, so the protocol model, its security proofs and the AArch64 semantics live in one logic, and Lean's kernel checks the refinement between them.

Mechanized computational cryptography. Game-based proofs [88, 22] have been mechanized in CertiCrypt [18], EasyCrypt [17], FCF [81], CryptHOL [19] and SSProve [1], among others, and CryptoVerif [31] automates such proofs. Our development builds on VCVio [97], a library for oracle computations in Lean. These frameworks handle the efficiency of reductions in different ways. Some leave it informal, and others restrict reductions syntactically. We write reductions in a small costed language and prove explicit cost bounds (§5.2.2).

Verified cryptographic code. Verified implementations of cryptographic *primitives* (as opposed to protocols) are complementary to our work, since they specify and verify functional correctness properties (and sometimes constant-time execution) of the primitives that we use in our verification of a cryptographic *protocol*. Many prior works have verified these primitives, at the C or assembly level, including HACL⋆ [108], Fiat Cryptography [51], Vale [34], Jasmin [4], the formally verified ML-KEM in `libcrux` [44], and the s2n-bignum and AWS-LC proofs in HOL Light [8]. Computational security proofs of cryptographic primitives have been carried down to assembly through verified compilers [5, 6, 3]. To our knowledge, ours are the first proofs of protocol-level security properties for the unmodified code of a widely deployed implementation of a cryptographic protocol.

Verifying code at the binary level. Our AArch64 semantics belongs to a line of work on formal ISA models, including Sail [14], which we test against, Islaris [85], which verifies machine code against such models, and LNSym [55], an Armv8 symbolic simulator in Lean 4. CompCert [73] and the translation validation of seL4 [87] address the gap between source and machine code by verifying the compiler or each compilation. Neither is available for Rust, and source-level Rust verifiers such as hax [25], Aeneas [58] and Verus [71] trust `rustc`, LLVM and a model of Rust's semantics, so we verify the compiler's output directly. Ironclad Apps [57] proved end-to-end security properties of applications at the assembly level, but for applications written in Dafny for verification.

AI for formal verification. Language models have been extensively used to generate formal proofs over the last few years, in both mathematics [82, 103, 62, 96] and software verification. Some efforts in the latter category generate proof annotations in verification-aware languages like Dafny, Verus and F⋆ [100, 92, 102, 39]. Others focus on automatically discharging proof obligations for interactive provers [94, 106, 53].

AI capabilities in this area are growing rapidly. In 2025, an agent completed a formalization of the strong prime number theorem in about 25,000 lines of Lean [75]. In September 2026, a multi-agent AI system formalized a proof of Fermat's Last Theorem in about 13 million lines [9]. In software verification, AI systems have been primarily evaluated on the verification of small individual programs [93, 104], though there is early work on repository-level verification [101].

A number of recent efforts, carried out concurrently with this effort, have highlighted the ability of AI to scale up verification to cover compiled machine code. This includes MachCSL, which verifies the xv6 OS kernel against RISC-V privileged hardware semantics [65], verification of open-source fuzz targets compiled to RISC-V [11], and Zellic's work on verifying glibc's startup on x86 [105]. These efforts largely target generic properties, such as crash freedom, or functional correctness. By contrast, our work shows that verifying security of a complex deployed system requires sophisticated specifications, which are substantial engineering projects in their own right. This is why AlphaProof Code supports AI-assisted specification discovery as well as autonomous proof.

9. Discussion

We have introduced AlphaProof Code, a framework in which humans and AI agents collaborate on the formal specification and verification of software, and used it to prove the security of `libsignal`, the Signal protocol's reference implementation, at the machine-code level. Our security proofs cover both the symbolic and the computational models. While there is a long line of prior work on formal verification of cryptographic protocols [32, 76, 17, 43, 28, 40, 27, 67], ours are the first proofs of protocol-level security properties for the unmodified code of a widely deployed implementation of a cryptographic protocol.

Our results can be extended in at least three directions. First, the theorems can be strengthened. Our computational theorem is classical, and our ratchet theorems let the attacker choose SPQR's keys, so neither proves the post-quantum healing that SPQR is designed to provide. The model choices of §5.2.5 and §5.1.6, such as a single session per handshake and fixed message lengths, can be relaxed. Second, the trusted base can be reduced. The two output functions and a few library routines are trusted contracts; verifying them would move the trust boundary down to the system calls. The low-level cryptographic routines have idealized contracts, which could be connected to verified implementations [108, 44]. Third, the scope can be expanded beyond one-to-one sessions, for example, to Signal's group messaging, sealed sender and server-side protocols.

More generally, our work is one of several recent efforts [11, 65] that point to a shift in the level of abstraction at which code is verified. The attraction of machine-code-level verification is that one needs to trust less—there is no need to model the semantics of high-level languages like Rust and C++ or assume that the compiler is correct. However, proofs at this level are long; when humans wrote almost all formal proofs, they were impractical beyond restricted domains. AI changes this equation by drastically reducing the labor cost of formal proof. That said, verifying at the source level remains a worthwhile way to reduce the compute cost of proof. For example, a substantial fraction of our proofs concerns registers, stack frames and memory layout, which source-level abstractions would hide. Extending our methods to higher-level languages would lower this cost, at the price of trusting the compiler and the language semantics, or of verifying them as well [73, 87].

One lesson is that while proofs are now much cheaper, specifications remain hard. *The specification of a nontrivial software artifact is itself a nontrivial software artifact.* Developing the software engineering discipline around formal specification is likely to become a fruitful area of research—conceptual frameworks, interfaces, and workflows through which humans and agents develop and validate a formal definition of what correctness and security mean. The process of §7, in which agents draft, review and test specifications under human direction, is a glimpse of how humans and AI might share this work. Like many of the most important problems in software engineering, this is a socio-technical challenge. Specifications of widely deployed systems need to be agreed upon by the communities that maintain, analyze and rely on them. Complex specifications, such as ours, will take many rounds of open review and refinement to settle. We are open-sourcing all Lean specifications and proofs, to facilitate such a process.

In the longer run, we expect AI-driven specification and verification to be tightly coupled with AI-driven coding. Today, verification happens after the code is written, if at all. Once every piece of code comes with a formal specification, and AI proofs are cheap, every change to the code can be verified before it is merged. AlphaProof Code extends naturally to this setting. It treats the specification as a living artifact, revised through reviewed amendments when proofs reveal that it is wrong (§2.3); code changes can be handled the same way, with groups of prover agents repairing the proofs that are affected when the code and (in many cases) the specification change. We are keen to use this workflow to verify new releases of `libsignal` before they ship, and also more broadly, as

well. The coupling of code generation, specification and verification through AI could at last make machine-checked proofs of correctness and security routine for critical systems.

Acknowledgments

We thank Eric Wieser for help with open-sourcing our code, Owen Jones for program management support, and Martín Abadi, Christopher Henson, Katerina Hristova, Henryk Michalewski, Jonathan Protzenko and George Tsoukalas for helpful discussions. We thank Signal's Rolfe Schmidt, Michael Rosenberg, and Ehren Kret for their feedback on our specifications. We also thank Tulsee Doshi, Four Flynn, Kate Olszewska, Alicia Huang, Andy Liu, Benoit Schillings, Raluca Ada Popa, Melvin Johnson, Koray Kavukcuoglu, Demis Hassabis, and Sundar Pichai for their support of the project.

References

[1] Carmine Abate, Philipp G. Haselwarter, Exequiel Rivas, Antoine Van Muylder, Théo Winterhalter, Cătălin Hriţcu, Kenji Maillard, and Bas Spitters. SSProve: A foundational framework for modular cryptographic proofs in Coq. In *IEEE Computer Security Foundations Symposium (CSF)*, pages 1–15. IEEE, 2021.

[2] Mihhail Aizatulin, Andrew D. Gordon, and Jan Jürjens. Extracting and verifying cryptographic models from C protocol code by symbolic execution. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS)*, pages 331–340. ACM, 2011. doi:10.1145/2046707.2046745.

[3] José Bacelar Almeida, Santiago Arranz-Olmos, Manuel Barbosa, Gilles Barthe, François Dupressoir, Benjamin Grégoire, Vincent Laporte, Jean-Christophe Léchenet, Cameron Low, Tiago Oliveira, Hugo Pacheco, Miguel Quaresma, Peter Schwabe, and Pierre-Yves Strub. Formally verifying Kyber – Episode V: Machine-checked IND-CCA security and correctness of ML-KEM in EasyCrypt. In *Advances in Cryptology – CRYPTO 2024*, volume 14921 of *Lecture Notes in Computer Science*, pages 384–421. Springer, 2024. doi:10.1007/978-3-031-68379-4_12.

[4] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. Jasmin: High-assurance and high-speed cryptography. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1807–1823. ACM, 2017.

[5] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, and François Dupressoir. Certified computer-aided cryptography: Efficient provably secure machine code from high-level implementations. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1217–1230. ACM, 2013. doi:10.1145/2508859.2516652.

[6] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Adrien Koutsos, Vincent Laporte, Tiago Oliveira, and Pierre-Yves Strub. The last mile: High-assurance and high-speed cryptographic implementations. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 965–982. IEEE, 2020. doi:10.1109/SP40000.2020.00028.

[7] Joël Alwen, Sandro Coretti, and Yevgeniy Dodis. The Double Ratchet: Security notions, proofs, and modularization for the Signal protocol. In *Advances in Cryptology – EUROCRYPT 2019*, volume 11476 of *Lecture Notes in Computer Science*, pages 129–158. Springer, 2019. doi:10.1007/978-3-030-17653-2_5.

[8] Amazon Web Services. s2n-bignum: verified big-number arithmetic for cryptography. https://github.com/awslabs/s2n-bignum, 2024.

[9] Anthropic. Formalizing Fermat's last theorem. https://www.anthropic.com/research/formalizing-fermats-last-theorem, September 2026. Lean code: https://github.com/anthropics/fermats-last-theorem.

[10] Anthropic. Project Glasswing: Securing critical software for the AI era. https://www.anthropic.com/news/project-glasswing, 2026.

[11] Anthropic and Theorem. LLMs are making formal software verification feasible at a much larger scale. https://www.anthropic.com/research/formal-verification, 2026.

[12] Arm Limited. Arm A-profile A64 instruction set architecture. Arm Architecture Exploration Tools, Release 2026-06 (machine-readable XML/ASL specification), 2026. URL: https://developer.arm.com/tools-and-software/exploration-tools.

[13] Arm Limited. Arm architecture reference manual for A-profile architecture. Document ARM DDI 0487, 2026. URL: https://developer.arm.com/documentation/ddi0487/latest/.

[14] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. ISA semantics for ARMv8-A, RISC-V, and CHERI-MIPS. *Proceedings of the ACM on Programming Languages*, 3(POPL):71:1–71:31, 2019.

[15] Linard Arquint, Felix A. Wolf, Joseph Lallemand, Ralf Sasse, Christoph Sprenger, Sven N. Wiesner, David Basin, and Peter Müller. Sound verification of security protocols: From design to interoperable implementations. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, pages 1077–1093. IEEE, 2023. doi:10.1109/SP46215.2023.10179325.

[16] Chris Bailey. nanoda_lib: An external type checker for Lean 4. https://github.com/ammkrn/nanoda_lib, 2026.

[17] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella-Béguelin. Computer-aided security proofs for the working cryptographer. In *Advances in Cryptology – CRYPTO 2011*, volume 6841 of *LNCS*, pages 71–90. Springer, 2011.

[18] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella-Béguelin. Formal certification of code-based cryptographic proofs. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 90–101. ACM, 2009.

[19] David A. Basin, Andreas Lochbihler, and S. Reza Sefidgar. CryptHOL: Game-based proofs in higher-order logic. *Journal of Cryptology*, 33(2):494–566, 2020.

[20] Ilan Beer, Shoham Ben-David, Cindy Eisner, and Yoav Rodeh. Efficient detection of vacuity in ACTL formulas. In *Computer Aided Verification (CAV)*, volume 1254 of *Lecture Notes in Computer Science*, pages 279–290. Springer, 1997.

[21] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *ACM Conference on Computer and Communications Security (CCS)*, pages 62–73, 1993.

[22] Mihir Bellare and Phillip Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In *Advances in Cryptology – EUROCRYPT 2006*, volume 4004 of *LNCS*, pages 409–426. Springer, 2006.

[23] Beneficial AI Foundation, Signal, and Lean FRO. Signal Shot: Formally verifying the Signal protocol and Signal app with Lean, 2026. Announced April 2026. URL: https://www.beneficialaifoundation.org/signal-shot.

[24] Karthikeyan Bhargavan, Abhishek Bichhawat, Quoc Huy Do, Pedram Hosseyni, Ralf Küsters, Guido Schmitz, and Tim Würtele. DY*: A modular symbolic verification framework for executable cryptographic protocol code. In *Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 523–542. IEEE, 2021. doi:10.1109/EuroSP51992.2021.00042.

[25] Karthikeyan Bhargavan, Maxime Buyse, Lucas Franceschino, Lasse Letager Hansen, Franziskus Kiefer, Jonas Schneider-Bensch, and Bas Spitters. hax: Verifying security-critical Rust software using multiple provers. In *Verified Software. Theories, Tools, and Experiments (VSTTE)*, volume 15525 of *Lecture Notes in Computer Science*, pages 96–119. Springer, 2025. doi:10.1007/978-3-031-86695-1_7.

[26] Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Stephen Tse. Verified interoperable implementations of security protocols. *ACM Transactions on Programming Languages and Systems*, 31(1):5:1–5:61, 2008. doi:10.1145/1452044.1452049.

[27] Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, and Pierre-Yves Strub. Implementing TLS with verified cryptographic security. In *IEEE Symposium on Security and Privacy (S&P)*, pages 445–459. IEEE, 2013.

[28] Karthikeyan Bhargavan, Charlie Jacomme, Franziskus Kiefer, and Rolfe Schmidt. Formal verification of the PQXDH post-quantum key agreement protocol for end-to-end secure messaging. In *Proceedings of the 33rd USENIX Security Symposium*, pages 469–486. USENIX Association, 2024. URL: https://www.usenix.org/conference/usenixsecurity24/presentation/bhargavan.

[29] Big Sleep Team. From Naptime to Big Sleep: Using large language models to catch vulnerabilities in real-world code. Google Project Zero blog, https://googleprojectzero.blogspot.com/2024/10/from-naptime-to-big-sleep.html, 2024.

[30] Bruno Blanchet. An efficient cryptographic protocol verifier based on Prolog rules. In *Proceedings of the 14th IEEE Computer Security Foundations Workshop (CSFW)*, pages 82–96. IEEE, 2001. doi:10.1109/CSFW.2001.930138.

[31] Bruno Blanchet. A computationally sound mechanized prover for security protocols. *IEEE Transactions on Dependable and Secure Computing*, 5(4):193–207, 2008.

[32] Bruno Blanchet. Modeling and verifying security protocols with the applied pi calculus and ProVerif. *Foundations and Trends in Privacy and Security*, 1(1–2):1–135, 2016. doi:10.1561/3300000004.

[33] Bruno Blanchet and Vincent Cheval. ProVerif: Automated symbolic protocol verification. https://bblanche.gitlabpages.inria.fr/proverif/. Accessed October 2026.

[34] Barry Bond, Chris Hawblitzel, Manos Kapritsos, K. Rustan M. Leino, Jacob R. Lorch, Bryan Parno, Ashay Rane, Srinath Setty, and Laure Thompson. Vale: Verifying high-performance cryptographic assembly code. In *USENIX Security Symposium*, pages 917–934. USENIX Association, 2017. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/bond.

[35] Jacqueline Brendel, Cas Cremers, Dennis Jackson, and Mang Zhao. The provable security of Ed25519: Theory and practice. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, pages 1659–1676. IEEE, 2021. doi:10.1109/SP40001.2021.00042.

[36] Mario Carneiro. The type theory of Lean. Master's thesis, Carnegie Mellon University, April 2019. https://github.com/digama0/lean-type-theory.

[37] Mario Carneiro. Lean4Lean: Verifying a typechecker for Lean, in Lean, 2024. arXiv:2403.14064.

[38] Sagar Chaki and Anupam Datta. ASPIER: An automated framework for verifying security protocol implementations. In *Proceedings of the 22nd IEEE Computer Security Foundations Symposium (CSF)*, pages 172–185. IEEE, 2009. doi:10.1109/CSF.2009.20.

[39] Saikat Chakraborty, Gabriel Ebner, Siddharth Bhat, Sarah Fakhoury, Sakina Fatima, Shuvendu Lahiri, and Nikhil Swamy. Towards neural synthesis for SMT-assisted proof-oriented programming. In *International Conference on Software Engineering (ICSE)*, 2025. arXiv:2405.01787.

[40] Vincent Cheval, Charlie Jacomme, and Jessica Richards. Automated formal analysis of Signal's Double Ratchet: Attacks, fixes and security proofs. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, pages 1522–1538. IEEE, 2026. Full version: Cryptology ePrint Archive, Paper 2026/727. URL: https://eprint.iacr.org/2026/727.

[41] Andrey Chudnov, Nathan Collins, Byron Cook, Joey Dodds, Brian Huffman, Colm MacCárthaigh, Stephen Magill, Eric Mertens, Eric Mullen, Serdar Tasiran, Aaron Tomb, and Eddy Westbrook. Continuous formal verification of Amazon s2n. In *Computer Aided Verification (CAV)*, volume 10982 of *Lecture Notes in Computer Science*, pages 430–446. Springer, 2018.

[42] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. A formal security analysis of the Signal messaging protocol. In *Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 451–466. IEEE, 2017.

[43] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. A formal security analysis of the Signal messaging protocol. *Journal of Cryptology*, 33(4):1914–1983, 2020. doi:10.1007/s00145-020-09360-1.

[44] Cryspen. libcrux: The formally verified crypto library. https://github.com/cryspen/libcrux, 2026.

[45] Leonardo de Moura and Sebastian Ullrich. The Lean 4 theorem prover and programming language. In *Automated Deduction – CADE 28*, volume 12699 of *LNCS*, pages 625–635. Springer, 2021.

[46] Jean Paul Degabriele, Anja Lehmann, Kenneth G. Paterson, Nigel P. Smart, and Mario Strefler. On the joint security of encryption and signature in EMV. In *Topics in Cryptology – CT-RSA 2012*, volume 7178 of *Lecture Notes in Computer Science*, pages 116–135. Springer, 2012. doi:10.1007/978-3-642-27954-6_8.

[47] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.

[48] Yevgeniy Dodis, Daniel Jost, Shuichi Katsumata, Thomas Prest, and Rolfe Schmidt. Triple ratchet: A bandwidth efficient hybrid-secure Signal protocol. In *Advances in Cryptology – EUROCRYPT 2025*, volume 15608 of *Lecture Notes in Computer Science*, pages 302–331. Springer, 2025.

[49] Danny Dolev and Andrew C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983. doi:10.1109/TIT.1983.1056650.

[50] François Dupressoir, Andrew D. Gordon, Jan Jürjens, and David A. Naumann. Guiding a general-purpose C verifier to prove cryptographic protocols. In *Proceedings of the 24th IEEE Computer Security Foundations Symposium (CSF)*, pages 3–17. IEEE, 2011. doi:10.1109/CSF.2011.8.

[51] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. Simple high-level code for cryptographic arithmetic – with proofs, without compromises. In *IEEE Symposium on Security and Privacy (S&P)*, pages 1202–1219. IEEE, 2019.

[52] Rune Fiedler and Felix Günther. Security analysis of Signal's PQXDH handshake. In *Public-Key Cryptography – PKC 2025*, volume 15675 of *Lecture Notes in Computer Science*, pages 137–169. Springer, 2025. Full version: Cryptology ePrint Archive, Paper 2024/702. URL: https://eprint.iacr.org/2024/702.

[53] Emily First, Markus N. Rabe, Talia Ringer, and Yuriy Brun. Baldur: Whole-proof generation and repair with large language models. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 1229–1241. ACM, 2023.

[54] Robert W. Floyd. Assigning meanings to programs. In *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages 19–32. American Mathematical Society, 1967.

[55] Shilpi Goel and LNSym Contributors. LNSym: An Armv8 native-code symbolic simulator in Lean. https://github.com/leanprover/LNSym, 2024.

[56] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28(2):270–299, 1984.

[57] Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. Ironclad apps: End-to-end security via automated full-system verification. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 165–181. USENIX Association, 2014.

[58] Son Ho and Jonathan Protzenko. Aeneas: Rust verification by functional translation. *Proceedings of the ACM on Programming Languages*, 6(ICFP):711–741, 2022. doi:10.1145/3547647.

[59] Son Ho, Jonathan Protzenko, Abhishek Bichhawat, and Karthikeyan Bhargavan. Noise*: A library of verified high-performance secure channel protocol implementations. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, pages 107–124. IEEE, 2022. doi:10.1109/SP46214.2022.9833621.

[60] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969. doi:10.1145/363235.363259.

[61] C. A. R. Hoare. The verifying compiler: A grand challenge for computing research. *Journal of the ACM*, 50(1):63–69, 2003.

[62] Thomas Hubert, Rishi Mehta, Laurent Sartran, Miklós Z Horváth, Goran Žužić, Eric Wieser, Aja Huang, Julian Schrittwieser, Yannick Schroecker, Hussain Masoom, et al. Olympiad-level formal mathematical reasoning with reinforcement learning. *Nature*, 651(8106):607–613, 2026.

[63] Simon Jeanteur, Lorenzo Veronese, Magdalena Solitro, and Matteo Maffei. LeanDY: Type-based and trace-based symbolic protocol verification in Lean. arXiv:2607.03406, 2026. URL: https://arxiv.org/abs/2607.03406.

[64] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, 2011.

[65] M. Frans Kaashoek and Nickolai Zeldovich. Extending concurrent separation logic to the hardware level to verify the xv6 OS kernel on RISC-V with AI agents. arXiv:2609.04043, 2026. URL: https://arxiv.org/abs/2609.04043.

[66] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, pages 207–220. ACM, 2009. doi:10.1145/1629575.1629596.

[67] Nadim Kobeissi, Karthikeyan Bhargavan, and Bruno Blanchet. Automated verification for secure messaging protocols and their implementations: A symbolic and computational approach. In *Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 435–450. IEEE, 2017.

[68] Hugo Krawczyk and Pasi Eronen. HMAC-based extract-and-expand key derivation function (HKDF). RFC 5869, 2010. URL: https://www.rfc-editor.org/rfc/rfc5869.

[69] Ehren Kret and Rolfe Schmidt. The PQXDH key agreement protocol. Signal specification, https://signal.org/docs/specifications/pqxdh/, 2023.

[70] Orna Kupferman and Moshe Y. Vardi. Vacuity detection in temporal model checking. *International Journal on Software Tools for Technology Transfer*, 4(2):224–233, 2003.

[71] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. Verus: Verifying Rust programs using linear ghost types. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA1):286–315, 2023. doi:10.1145/3586037.

[72] Lean FRO. comparator: checking Lean proofs against a reference statement. https://github.com/leanprover/comparator, 2025.

[73] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[74] Moxie Marlinspike. Advanced cryptographic ratcheting. Signal blog, https://signal.org/blog/advanced-ratcheting/, 2013.

[75] Math, Inc. The strong prime number theorem: An AI-generated Lean formalization. https://github.com/math-inc/strongpnt, September 2025. Produced by the Gauss autoformalization agent, https://www.math.inc/gauss.

[76] Simon Meier, Benedikt Schmidt, Cas Cremers, and David Basin. The TAMARIN prover for the symbolic analysis of security protocols. In *Computer Aided Verification (CAV)*, volume 8044 of *Lecture Notes in Computer Science*, pages 696–701. Springer, 2013. doi:10.1007/978-3-642-39799-8_48.

[77] OpenAI. Introducing Aardvark: OpenAI's agentic security researcher. https://openai.com/index/introducing-aardvark/, 2025.

[78] Davide Paglieri, Logan Cross, Tim Genewein, Joel Z. Leibo, Nenad Tomasev, and Alexander Sasha Vezhnevets. A case study on emergent cheating and whistleblowing in autonomous research swarms. arXiv:2609.04170, 2026. URL: https://arxiv.org/abs/2609.04170.

[79] Trevor Perrin. The XEdDSA and VXEdDSA signature schemes. Signal specification, https://signal.org/docs/specifications/xeddsa/, 2016.

[80] Trevor Perrin and Moxie Marlinspike. The double ratchet algorithm. Signal specification, https://signal.org/docs/specifications/doubleratchet/, 2016.

[81] Adam Petcher and Greg Morrisett. The foundational cryptography framework. In *Principles of Security and Trust (POST)*, volume 9036 of *LNCS*, pages 53–72. Springer, 2015.

[82] Stanislas Polu and Ilya Sutskever. Generative language modeling for automated theorem proving. arXiv:2009.03393, 2020. URL: https://arxiv.org/abs/2009.03393.

[83] Jonathan Protzenko, Benjamin Beurdouche, Denis Merigoux, and Karthikeyan Bhargavan. Formally verified cryptographic web applications in WebAssembly. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, pages 1256–1274. IEEE, 2019. doi:10.1109/SP.2019.00064.

[84] Alastair Reid. Trustworthy specifications of ARM v8-A and v8-M system level architecture. In *Proceedings of Formal Methods in Computer-Aided Design (FMCAD)*, pages 161–168. FMCAD Inc., 2016. doi:10.1109/FMCAD.2016.7886675.

[85] Michael Sammler, Angus Hammond, Rodolphe Lepigre, Brian Campbell, Jean Pichon-Pharabod, Derek Dreyer, Deepak Garg, and Peter Sewell. Islaris: Verification of machine code against authoritative ISA semantics. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 825–840. ACM, 2022.

[86] Benedikt Schmidt, Simon Meier, Cas Cremers, and David Basin. Automated analysis of Diffie-Hellman protocols and advanced security properties. In *Proceedings of the 25th IEEE Computer Security Foundations Symposium (CSF)*, pages 78–94. IEEE, 2012. doi:10.1109/CSF.2012.25.

[87] Thomas Sewell, Magnus O. Myreen, and Gerwin Klein. Translation validation for a verified OS kernel. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 471–482. ACM, 2013.

[88] Victor Shoup. Sequences of games: A tool for taming complexity in security proofs. Cryptology ePrint Archive, Paper 2004/332, 2004. URL: https://eprint.iacr.org/2004/332.

[89] Signal. The sparse post-quantum ratchet (SPQR). Signal specification and the `spqr` crate, 2025. URL: https://signal.org/blog/spqr/.

[90] Signal and Cryspen. SparsePostQuantumRatchet: Formal verification of the SPQR Rust implementation with hax and F*. https://github.com/signalapp/SparsePostQuantumRatchet, 2025.

[91] Signal Messenger. libsignal. https://github.com/signalapp/libsignal, 2026. Version 0.105.4.

[92] Chuyue Sun, Ying Sheng, Oded Padon, and Clark Barrett. Clover: Closed-loop verifiable code generation. In *International Symposium on AI Verification (SAIV)*, Lecture Notes in Computer Science, pages 134–155. Springer, 2024. arXiv:2310.17807.

[93] Amitayush Thakur, Jasper Lee, George Tsoukalas, Meghana Sistla, Matthew Zhao, Stefan Zetzsche, Greg Durrett, Yisong Yue, and Swarat Chaudhuri. CLEVER: A curated benchmark for formally verified code generation. *Advances in Neural Information Processing Systems*, 38, 2025.

[94] Amitayush Thakur, George Tsoukalas, Yeming Wen, Jimmy Xin, and Swarat Chaudhuri. An in-context learning agent for formal theorem-proving. In *Conference on Language Modeling (COLM)*, 2024. arXiv:2310.04353.

[95] Erik Thormarker. On using the same key pair for Ed25519 and an X25519 based KEM. Cryptology ePrint Archive, Paper 2021/509, 2021. URL: https://eprint.iacr.org/2021/509.

[96] George Tsoukalas, Anton Kovsharov, Sergey Shirobokov, Anja Surina, Moritz Firsching, Gergely Bérczi, Francisco JR Ruiz, Arun Suggala, Adam Zsolt Wagner, Eric Wieser, et al. Advancing mathematics research with AI-driven formal proof search. *Science*, 394(6820):234–239, 2026.

[97] Devon Tuma and Nicholas Hopper. VCVio: A formally verified forking lemma and Fiat-Shamir transform, via a flexible and expressive oracle representation. Cryptology ePrint Archive, Paper 2024/1819, 2024. URL: https://eprint.iacr.org/2024/1819.

[98] Alan M. Turing. Checking a large routine. In *Report of a Conference on High Speed Automatic Calculating-Machines*, pages 67–69. University Mathematical Laboratory, Cambridge, 1949.

[99] Théophile Wallez and Cas Cremers. Bob DyLean: A framework for the symbolic analysis of cryptographic protocols in Lean. Cryptology ePrint Archive, Paper 2026/1493, 2026. URL: https://eprint.iacr.org/2026/1493.

[100] Haoze Wu, Clark Barrett, and Nina Narodytska. Lemur: Integrating large language models in automated program verification. In *International Conference on Learning Representations (ICLR)*, 2024. arXiv:2310.04870.

[101] Yutong Xin, Qiaochu Chen, Greg Durrett, and Işil Dillig. VeriSoftBench: Repository-scale formal verification benchmarks for Lean. *arXiv preprint arXiv:2602.18307*, 2026.

[102] Chenyuan Yang, Xuheng Li, Md Rakib Hossain Misu, Jianan Yao, Weidong Cui, Yeyun Gong, Chris Hawblitzel, Shuvendu Lahiri, Jacob R. Lorch, Shuai Lu, Fan Yang, Ziqiao Zhou, and Shan Lu. AutoVerus: Automated proof generation for Rust code. arXiv:2409.13082; OOPSLA, 2024.

[103] Kaiyu Yang, Aidan M. Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan Prenger, and Anima Anandkumar. LeanDojo: Theorem proving with retrieval-augmented language models. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 36, pages 21573–21612, 2023.

[104] Zhe Ye, Zhengxu Yan, Jingxuan He, Timothe Kasriel, Kaiyu Yang, and Dawn Song. Verina: Benchmarking verifiable code generation. In *International Conference on Learning Representations*, pages 38933–38972, 2026.

[105] Zellic and V12. Announcing Zellic's and V12's work on x86 binary formal verification. https://www.zellic.io/blog/x86-formal-verification-intro/, 2026.

[106] Lichen Zhang, Shuai Lu, and Nan Duan. Selene: Pioneering automated proof in software verification. In *Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 1776–1789, 2024. arXiv:2401.07663.

[107] Ziqian Zhong, Aditi Raghunathan, and Nicholas Carlini. ImpossibleBench: Measuring LLMs' propensity of exploiting test cases. arXiv:2510.20270, 2025. URL: https://arxiv.org/abs/2510.20270.

[108] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. HACL*: A verified modern cryptographic library. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1789–1806. ACM, 2017.

A. The Signal Protocol in Detail

This appendix gives the details of PQXDH and the Triple Ratchet that §3 omits (Figure 8).

PQXDH. Each party has a long-term X25519 identity key IK. Bob uploads to the server a *prekey bundle*: a signed prekey SPK_B and a Kyber1024 prekey $PQPK_B$ (single-use, or a reusable *last-resort* key), both signed with IK_B (XEdDSA), and optionally single-use *one-time prekeys* OPK_B. Alice fetches the bundle, verifies the signatures, generates an ephemeral key EK_A and computes $DH_1 = \mathsf{DH}(IK_A, SPK_B)$, $DH_2 = \mathsf{DH}(EK_A, IK_B)$, $DH_3 = \mathsf{DH}(EK_A, SPK_B)$ and, if a one-time prekey was used, $DH_4 = \mathsf{DH}(EK_A, OPK_B)$. She encapsulates a secret ss to $PQPK_B$, obtaining a ciphertext ct, and derives the session secret SK with HKDF from the DH values and ss. The DH values implicitly authenticate the parties, since only the holders of the identity keys can compute them, and ss protects SK against an adversary who records the handshake and later breaks Diffie-Hellman. Her first message, a *PreKeySignalMessage*, carries IK_A, EK_A, ct and the prekey identifiers together with her first encrypted message. Bob recomputes SK and deletes the one-time prekey.

The Triple Ratchet. SK is split into a root key, a chain key and a seed for SPQR. In the Double Ratchet [80], the *symmetric ratchet* derives each message key and the next chain key from the current chain key with HMAC-SHA256, so old keys can be deleted (*forward secrecy*). Each message carries the sender's ratchet public key, and a party that sees a new one mixes a fresh DH output into the root key, so an adversary who has compromised a party's state loses access after the next round trip, provided it does not interfere with that round trip (*post-compromise security*). This healing holds only against classical adversaries, so the Sparse Post-Quantum Ratchet (SPQR) [89] runs an ML-KEM-768 exchange alongside, with keys and ciphertexts split into chunks carried in ordinary messages; each completed exchange yields an epoch key from which SPQR derives a key per message. The key that

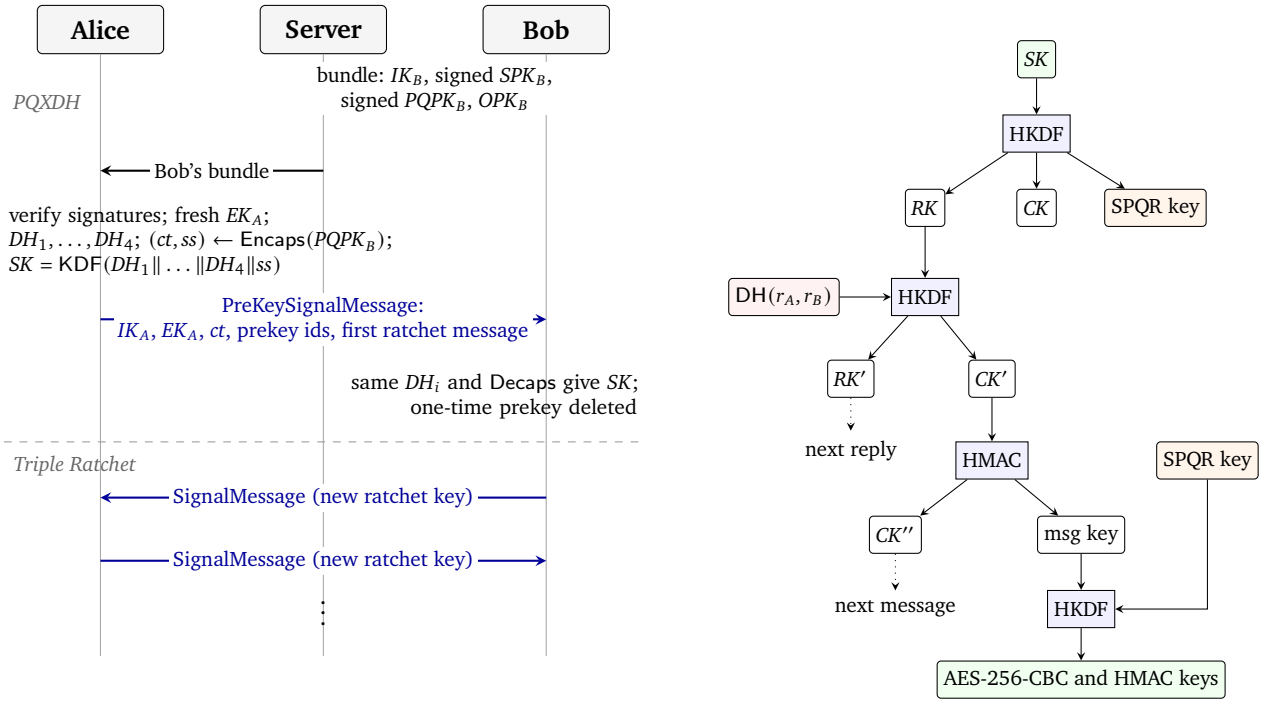


Figure 8 | The Signal protocol as implemented by `libsignal`, in detail. Left: Bob uploads his prekey bundle to the server; Alice fetches it, runs PQXDH, and sends a prekey message that carries the handshake values together with her first encrypted message. She keeps sending prekey messages until Bob replies; after that, the parties exchange ordinary Triple Ratchet messages. Right, simplified: the handshake secret *SK* yields a root key, a chain key and an SPQR key. Each reply with a new ratchet key mixes a fresh Diffie-Hellman output into the root key; each message advances the symmetric chain; and each message key is mixed with a key from SPQR before it is used to encrypt and authenticate the message.

protects a message is derived from both the Double Ratchet's and SPQR's message keys. The message is encrypted with AES-256-CBC and authenticated with HMAC-SHA256 truncated to 8 bytes, over both identity keys and the protobuf-encoded *SignalMessage*, which carries the sender's ratchet key, counters for out-of-order delivery, the SPQR chunk and the ciphertext. The receiver keeps a bounded number of keys of skipped messages.

B. Symbolic Proofs

This appendix completes §5.1: the composed model as a transition system (§B.1), the hardware events (§B.2), the concretization function (§B.3), explanations (§B.4), the collision clause (§B.5), the security statement and the scope of the binary theorem (§B.6, §B.7), and the preservation proof (§B.8).

B.1. The Composed Model

Each of the four `libsignal` entry points corresponds to one atomic transition in DyLean's `Traceful` monad (so the DH and SPQR ratchets execute atomically inside `messageDecrypt`, never exposing a half-ratcheted state). Transitions read from and append to a single global append-only trace (`ExecTrace`), recording fresh randomness (`genRand`), network transmissions (`sendMessage`), immutable session-state snapshots referenced by trace handles (`storeLocalState`), and pro-

tocol events (`logEvent`). Compromises, exposures and breaks are protocol events (such as `CompromisedIK`, `ExposedRK` and `BrokenDH`); a compromise or exposure then sends the leaked secret on the network. Table 3 summarizes all 23 transitions of the composed state machine (`ReachabilityConfig`), which store long-term and prekey secrets (`PQXDHState`) as well as session snapshots (`TRState`) on the trace; any transition may fire at any time on attacker-chosen arguments whose guards hold.

B.2. Hardware Events

A hardware trace is a list of events (`CliEvents`). A run event holds the party, the command, the run's input, its `coins`, and its exit code with its standard output. The attacker may invert a key only after the matching break, and only a key that an honest party generated: a key of the setup, or one that a successful run drew from its `coins`. Table 4 lists every event.

B.3. Concretization

γ (§5.1.4) is defined by recursion on terms: each function symbol f comes with a function on byte strings, and $\gamma(f(t_1,\dots,t_k))$ applies it to $\gamma(t_1),\dots,\gamma(t_k)$.

Literals stand for their own bytes, `concat` appends, `trunc8` keeps the first 8 bytes, and the wire-message symbols give `libsignal`'s protobuf encoding. The cryptographic symbols use an *interpretation* $\mathcal{I}$, which gives each symbol's result on its argument bytes. `dhPk` and `dh` are X25519, `symEnc` is AES-256-CBC, `Encap` is Kyber1024 encapsulation, and `kdfExtract` is HMAC-SHA256, built in Lean from the SHA-256 compression function. Among the laws assumed of $\mathcal{I}$ is Diffie-Hellman commutativity: `dh` sorts its two secrets, so both parties compute the same shared-secret term, and commutativity gives them the same bytes. The remaining laws state that X25519 public keys are canonical and never low-order, that agreement reads only the u-coordinate, ignores torsion points and is zero exactly on low-order points, that the binary's torsion check rejects a public key shifted by a torsion point (without this law the theorem is false, since the MAC does not cover the base key), that AES-CBC decryption inverts encryption and has no junk (a ciphertext that decrypts is the encryption of its plaintext), and that X25519 outputs and `libcrux` HMAC tags are 32 bytes long. These laws are trusted; we prove only that they are jointly satisfiable.

Two more parameters complete γ. Some symbols are *opaque*: the SPQR operations, Kyber1024 public keys, decapsulation of a ciphertext that is not an honest encapsulation, and XEdDSA. Their bytes come from fixed byte functions S, which must agree with the setup: they give the setup's signatures and Kyber public key their provisioned bytes, and decapsulation under Bob's Kyber prekey agrees with $\mathcal{I}$. Nothing else is assumed about them. A per-trace *ghost* g gives the bytes of each fresh value: those of the piece of a run's `coins`, or of the plaintext of a `send` run, that it is matched with, or, for the setup's keys, the setup's bytes. It also gives the bytes of each wire message's `addresses` field, which has no model term. The ghost is the only part of γ that depends on the execution.

B.4. Explanations

Each transition along a model trace (§5.1.1) corresponds to at most one event from Table 4, carrying symbolic terms rather than bytes: for example, encrypting a message corresponds to a `send` run whose input is a fresh plaintext term, whose `coins` are fresh coins, and whose output is a wire-message term; compromising a secret corresponds to a `leak`; and a rejected run leaves the model state unchanged. A model trace *matches* a hardware trace τ if these events line up with τ's events one to one, in order, and each byte field of a hardware event equals γ applied to the corresponding model term(s) under

Transition	Effect on the global trace (`ExecTrace`)
Network, PKI, and prekey upload	
`Network`	Attacker sends any term derivable from its current knowledge onto the network.
`PQXDH.registerIdentity`	Draws a fresh identity secret (once per party), logs honest PKI registration, sends public and verification keys, and stores `PQXDHState`.
`PQXDH.registerCorruptIdentity`	Reads an attacker-known secret (once per party) and logs dishonest PKI registration and `CompromisedIK`; stores no local state.
`PQXDH.genSignedPrekey`	Reads identity key from `PQXDHState`, draws a signed prekey, logs upload, stores `PQXDHState`, and sends public key and signature.
`PQXDH.genPQPrekey`	Reads identity key from `PQXDHState`, draws a Kyber1024 prekey, logs upload, stores `PQXDHState`, and sends public key and signature.
`PQXDH.genOneTimePrekey`	Draws an unsigned one-time prekey, logs upload, stores `PQXDHState`, and sends the public key.
Session start and Triple Ratchet messaging	
`aliceStart`	Verifies Bob's bundle (`process_prekey_bundle`), derives PQXDH keys, sends base key and KEM ciphertext, logs `AliceDone`, and stores initial `TRState`.
`bobStart`	Atomically runs PQXDH and decrypts the opening message (`message_decrypt_prekey`); marks any one-time prekey used, logs `BobDone` and skipped seeds, and stores `TRState`.
`TripleRatchet.messageEncrypt`	Loads current `TRState`, draws plaintext and SPQR coins, publishes SPQR key, advances sender chain, stores updated `TRState`, and sends wire message.
`TripleRatchet.messageEncryptAbort`	Loads current `TRState`, draws a fresh plaintext on the trace, and aborts without updating state or sending a message.
`TripleRatchet.messageDecrypt`	Loads current `TRState` (DH-ratcheting on a new key), publishes SPQR key, verifies and decrypts message, logs skipped seeds and `Received`, and updates `TRState`.
`TripleRatchet.messageDecryptSkipped`	Decrypts an out-of-order message on current `TRState` using an unused skipped seed, publishes SPQR key, marks the seed used, and updates SPQR in `TRState`.
Adversarial compromises and cryptographic breaks	
`TripleRatchet.exposeKey`	Leaks one key (root, DH ratchet, sending chain, or receiving chain) from any stored `TRState` to the network and logs its exposure event.
`TripleRatchet.exposeSpqr`	Leaks the SPQR state from any stored `TRState` to the network and logs `ExposedSpqr`.
`TripleRatchet.exposeSkipped`	Leaks an unused skipped message-key seed to the network and logs `ExposedMK`.
`PQXDH.compromiseIK`	Leaks a stored identity secret from `PQXDHState` to the network and logs `CompromisedIK`.
`PQXDH.compromiseSPK`	Leaks a stored signed-prekey secret from `PQXDHState` to the network and logs `CompromisedSPK`.
`PQXDH.compromisePQSPK`	Leaks a stored Kyber-prekey secret from `PQXDHState` to the network and logs `CompromisedPQSPK`.
`PQXDH.compromiseOPK`	Leaks an unused one-time-prekey secret from `PQXDHState` to the network and logs `CompromisedOPK`.
`PQXDH.breakDH`	Logs `BrokenDH`, enabling quantum breaks of Diffie-Hellman and XEdDSA signatures from this point on.
`PQXDH.breakKEM`	Logs `BrokenKEM`, enabling breaks of the KEM from this point on.
`PQXDH.invertDH`	After `BrokenDH`, reads an X25519 public key from the network and sends its discrete-log secret.
`PQXDH.invertKEM`	After `BrokenKEM`, reads a KEM public key from the network and sends its secret key.

Table 3 | State transitions of the composed symbolic model (`ReachabilityConfig`). Each transition executes atomically in DyLean's `Traceful` monad; the four `TripleRatchet` messaging transitions act on the party's current `TRState` snapshot for its role (so each party has at most one active session per role), whereas `exposeKey` and `exposeSpqr` apply to any stored snapshot.

Event	Records	Bytes	Matched by
Run events: one run of `libsignal-cli`			
`init`	a run of `init` (`process_prekey_bundle`)	the bundle it read	Alice's session start
`send`	a run of `send` (`message_encrypt`)	the plaintext it read and the wire message it printed	an encryption
`recv`	a run of `recv` (`message_decrypt`)	the wire message it read and the plaintext it printed	Bob's session start or a decryption
Environment events: actions of the attacker between runs			
`leak`	a compromise	a long-term or prekey secret, or a key in a party's current session record	a compromise or exposure
`breakDH`	a break of DH	—	the break
`breakKEM`	a break of the KEM	—	the break
`invertDH`	an inversion of an honest X25519 key	the public key and its secret	the inversion
`invertKEM`	an inversion of Bob's Kyber prekey	the public key and its secret	the inversion

Table 4 | The hardware-level events. Every run event also records the run's party, its `coins` and its exit code; a run that does not exit 0 is matched by no model transition. The last column gives the model transition that matches each event (§5.1.4). The attacker observes the bundles, the wire messages and the leaked or inverted secrets, which match network messages of the model trace, but not the plaintexts or the `coins`.

`libsignal`'s wire and key encodings.

A model trace *explains* τ if it is reachable in the model, matches τ, and agrees with the execution on details that the hardware events do not record. Because τ records only what goes into and out of each run, it omits how stored session records overwrite prior state from run to run, and how incoming bytes were decoded and routed internally. Pinning explanations to these unlogged details prevents the model trace from reusing replaced session state or logging handshakes the binary did not perform when we evaluate the security queries on it, and—in the collision clause below—rules out degenerate explanations of a trace prefix.

B.5. The Collision Clause

The collision clause of the refinement definition (§5.1.4) excuses only runs that no explanation can account for without a collision between two different terms, at least one built by a symbol assumed injective. No opaque symbol is assumed injective. Two further clauses concern the truncated MAC: a received message is also excused if its 8-byte MAC equals one that the receiver computes from a ratchet seed the attacker does not know (a guessed MAC), or one computed under a different seed. Three features constrain the collision case. First, S is chosen before the hardware trace, and opaque bytes are never assumed injective, so S cannot be tuned to create collisions. Second, for a received message the collision must occur under *every* explanation of the prefix that has not already read a colliding message (except in the Kyber case below), so the proof cannot pick an explanation that happens to collide. Third, a received message counts as a collision only if every term the attacker knows that the model would accept for it itself involves a collision: once the attacker has leaked or inverted a key, a message it forges with that key must be explained, not excused.

One further clause concerns Kyber: a decapsulation that disagrees with encapsulation is excused,

since no Kyber correctness law is assumed on $\mathcal{I}$; hardware traces in this case get no guarantee. The clause reads only the interpretation $\mathcal{I}$, the setup and the received ciphertext, never S, the ghost or the run, so it holds only if decapsulation under Bob's provisioned key disagrees with encapsulation on that ciphertext: the hooked Kyber implementation is incorrect, the provisioned key is malformed (key generation is not verified, and well-formedness checks only the public key embedded in the secret key), or the ciphertext is one of the decryption failures that Kyber1024 has with negligible probability. (Kyber's implicit rejection of a malformed ciphertext is modeled directly in the symbolic KEM by a secret-keyed rejection symbol, so it is explained rather than excused.)

B.6. The Security Statement

For initiator secrecy, for instance, the query in $Q(tr, K)$ says that if Alice completed a handshake in tr and one of its keys is in K, then one of that row's excuses of Table 5 holds on tr. The ten properties differ only in Q, where each is one conjunct: stated alone, each has exactly the shape of Theorem 1 with its own query in place of Q, and Theorem 1 implies all ten of these statements, with one common S, because Q is their conjunction. Table 5 states every query using the event shorthands defined next (for the events of Table 3).

A handshake query fixes the session's parties a (Alice) and b (Bob), its keys *ikA*, *ikB*, *spk*, *pqpk*, *opk* and outputs $keys = \{rk, ck, pqr\}$, and the index i of Alice's `AliceDone` or Bob's `BobDone` event; Done is either. IK_A, IK_B, SPK, PQ, OPK are the compromise events of the session's identity keys and prekeys, and DH, KEM the breaks. An event E alone means E is logged anywhere on tr; $E^{<i}$ and $E^{\leq i}$, before and at or before i. $\mathsf{Exp}(k)$ (`SessionStateExposed`) holds if block k itself was exposed from a session started with *keys*. For the ratchet, m is a used message seed with SPQR salt s; $ck \rightsquigarrow m$ and $rk \rightsquigarrow ck$ mean derivation by chain and root ratchet steps, and $rk \preceq rk'$ that rk' descends from rk. $C(x)$ (`CompromisedKey`) holds if x was exposed, or is a handshake's initial rk or ck and every handshake deriving it is excused by initiator or responder secrecy, or is an honest DH ratchet secret and DH holds.

B.7. Scope of the Binary Theorem

The hardware traces fix a narrower scenario than the model; on other traces the theorem says nothing. The main restrictions are as follows (the Lean statement adds technical conditions on file paths). Alice and Bob each run on one device, and each starts with the other's identity key pinned. Bob has one signed prekey, one reusable last-resort Kyber prekey and at most one one-time prekey; one-time Kyber prekeys are not covered. Only Alice runs `init`, and only when she has no session; only Bob receives prekey messages. The bundle Alice receives carries Bob's genuine identity key, signed prekey and Kyber prekey with their signatures (the unsigned one-time prekey is unconstrained), and received SPQR fields come from earlier honest messages. A leak reveals only Alice's identity secret, Bob's identity, signed-prekey and Kyber secrets, his one-time prekey while unused, and the keys of a party's current session. Inputs are below 2^{32} bytes (a prekey message's inner message below $2^{32} - 2048$), files below 2^{31} bytes, `coins` files exactly 192, 64 and 32 bytes for `init`, `send` and `recv`, and there are fewer than $2^{32} - 1$ sends. After Bob accepts a prekey message that does not match his current session, he makes no call until the environment deletes his session file. The theorem also assumes fresh coins (§5.1.5). Within this scenario the attacker schedules every run, controls the network, may delete a party's session file, and may leak secrets and break DH and the KEM, as in the model.

Query	If	then
Forward secrecy	m used, $K(m) \vee K(\mathit{cipherKey}(s,m))$	$C(m) \vee \exists ck{\rightsquigarrow}m.\ C(ck) \vee \exists rk{\rightsquigarrow}ck{\rightsquigarrow}m.\ C(rk)$
Post-compromise security	as above; x, y honest DH ratchet secrets, $(rk_h, ck_h) = \mathit{createChain}(rk_p, \mathit{dhPk}(y), x)$, $ck_h{\rightsquigarrow}m$ or $rk_h{\rightsquigarrow}ck{\rightsquigarrow}m$	first two as above; $\exists rk \succeq rk_h.\ rk{\rightsquigarrow}ck{\rightsquigarrow}m \wedge C(rk)$; or $(\exists rk \preceq rk_p.\ C(rk)) \wedge (C(x) \vee C(y))$
Initiator secrecy	$\mathsf{AliceDone}^i$, $k \in \mathit{keys}$, $K(k)$	$\mathsf{IK}_B^{<i} \vee (\mathsf{IK}_B \wedge \mathsf{SPK} \wedge (\mathsf{PQ} \vee \mathsf{KEM})) \vee \mathsf{DH}^{<i} \vee (\mathsf{DH} \wedge (\mathsf{PQ} \vee \mathsf{KEM})) \vee \mathsf{Exp}(k)$
Responder secrecy	$\mathsf{BobDone}^i$, $k \in \mathit{keys}$, $K(k)$	$\mathsf{Auth}_i \vee \big(\mathsf{IK}_B \wedge \mathsf{SPK} \wedge (\mathit{opk}{=}\bot \vee \mathsf{OPK}^{<i}) \wedge (\mathsf{PQ} \vee \mathsf{KEM})\big) \vee (\mathsf{DH} \wedge (\mathsf{PQ} \vee \mathsf{KEM})) \vee \mathsf{Exp}(k)$
Authentication	$\mathsf{BobDone}^i$	$\mathsf{Agree}_i \vee \mathsf{Auth}_i$
Peer auth.; KCI	$\mathsf{BobDone}^i$, $\neg\mathsf{Auth}_i$; for KCI also IK_B	Agree_i
Handshake FS	Done^i, $\neg\mathsf{IK}_A^{\le i}$, $\neg\mathsf{IK}_B^{\le i}$, $\neg\mathsf{SPK}$, $\neg\mathsf{PQ}$, $\neg\mathsf{DH}$, $\neg\mathsf{KEM}$	for each $k \in \mathit{keys}$: $\mathsf{Exp}(k) \vee \neg K(k)$
HNDL	Done^i, $\neg\mathsf{DH}^{\le i}$, $\neg\mathsf{KEM}$, $\neg\mathsf{IK}_A$, $\neg\mathsf{IK}_B$, $\neg\mathsf{SPK}$, $\neg\mathsf{PQ}$	as above
Session independence	Done^i, $\neg\mathsf{IK}_A$, $\neg\mathsf{IK}_B$, $\neg\mathsf{DH}$, $\neg\mathsf{KEM}$; b's prekeys compromised only if not spk, pqpk, $\cong \mathit{opk}$	as above

$\mathsf{Auth}_i := \mathsf{IK}_A^{<i} \vee \mathsf{SPK}^{<i} \vee \mathsf{DH}^{<i}$. Agree_i: some $\mathsf{AliceDone}^j$ by a, with any peer, $j < i$, the same pqpk, and spk, opk equal up to X25519 re-encoding ($\cong$).

Table 5 | The ten queries of $Q(tr, K)$ (`AllSecurityQueriesHold`), in the notation of Appendix B.6. Each row reads: for all sessions (or seeds) satisfying *If*, one disjunct of *then* holds.

B.8. Refinement Preservation

The preservation theorem is proved once, independently of the binary: for every predicate relating the hooks to $\mathcal{I}$ and every run relation, the ten model theorems and the refinement definition of §5.1.4 imply the statement of Theorem 1. The machine code enters only through the refinement theorem, so a new build of `libsignal-cli` needs a new refinement proof but no new security proof.

The proof has two steps. First, every explaining model trace is reachable, so each model theorem applies to it. Second, every term that the hardware attacker derives under an explanation is known to the model attacker on the explaining trace. Only this second step takes real work: each leaked or inverted secret matches a model step that reveals it, and two kinds of terms need model invariants showing that they already reached the attacker: the ephemeral key and Kyber ciphertext that Alice's binary resends in every prekey message while her prekey is pending, and Bob's one-time prekey.

The queries then transfer because of their shape. Each one either ignores the attacker's knowledge (authentication, peer authentication, KCI) or uses it only as a hypothesis ("if the attacker knows this key, then an excuse holds"), and every excuse is an event of the model trace, never a fact about knowledge. So an attacker that knows less cannot break a query. A query whose excuse read "the attacker knows k" would not transfer this way.

In the collision case, preservation still gives the ten queries on every explanation of the prefix before the colliding run, so a collision voids the guarantees only from that run on.

C. Computational Proofs

This appendix gives the precise game (§C.1), the cost semantics of reductions (§C.2), the four lemmas (§C.3), the full main theorem and its proof (§C.4), and the assumptions and restrictions in detail (§C.5, §C.6). The remaining subsections list the premises on the primitives and the terms of the single-session bound.

C.1. The Composed Game

The challenger holds every party's state (in the game on the executable, the party's directory) and plays the server that distributes identity keys and prekeys. It also supplies the coins of every run, except that the adversary chooses the coins of sends.

The game is written once, as a generic game $G(\Sigma)$ parameterized by a *protocol interface* Σ that supplies the handshake and ratchet steps. The oracles and the winning condition consult Σ only through these steps, for example to decide whether an injected message is accepted. We write $G(\Sigma^H_{\text{model}})$ for the game whose interface is the byte-level model with handshake H, and $G(\Sigma_{\text{bin}})$ for the game whose interface is runs of `libsignal-cli`. The advantage of A is $2\,|\Pr[A \text{ wins}] - \frac{1}{2}|$, written $\mathsf{Adv}^{\text{comp}}(A)$ in $G(\Sigma^H_{\text{model}})$ and $\mathsf{Adv}^{\text{bin}}(A)$ in $G(\Sigma_{\text{bin}})$. In the main text, $\mathsf{Adv}^{\text{comp}}$ refers to $H = H_{\text{lib}}$.

Handshake oracles. These are the oracles of the PQXDH key-indistinguishability game of [52]: registration of honest parties and of adversarial keys (whose secret keys the adversary must supply), generation of signed and one-time prekeys, initiation and acceptance, corruption of identity keys, reveal of session keys and of signed-prekey secrets, and, if the handshake models the session-key HKDF as a random oracle, queries to that oracle. The handshake itself is a parameter H.

Promotion. The test query of the key-indistinguishability game is withheld from the adversary. Instead, the adversary may promote an initiator session *sid* by calling $\mathsf{startDR}(sid)$. The oracle checks that *sid* was initiated through the game and not yet promoted. It checks that *sid* used, byte for byte, the identity key, signed prekey, Kyber prekey and (if any) one-time prekey that a registered party published, with matching prekey identifiers, and that no earlier instance used the same ephemeral key. If so, it initializes the next ratchet instance, including SPQR, for both parties with the *real* session key. Finally, it marks the responder's signed prekey used in the handshake as revealed in the handshake game; the secret goes to the ratchet, not to the adversary.

Ratchet oracles. Each ratchet instance has the oracles of the ACD19 secure-messaging game [7]: transmit, challenge (m_0, m_1), deliver, inject (a forgery attempt) and corruption of either party's state. The adversary wins if it guesses the challenge bit b or gets a non-trivial forgery accepted, subject to ACD19's cleanness conditions and the further restrictions of §C.6. If at the end the handshake part of the execution violates the freshness condition of [52], the game outputs a random bit.

SPQR. The ratchet in each session is `libsignal`'s Triple Ratchet: the Double Ratchet together with SPQR. SPQR carries ML-KEM-768 exchanges in chunks inside the message headers and, as they complete, outputs fresh 32-byte keys. Each message key is derived from the Double Ratchet's chain key with the current SPQR key as the HKDF salt. SPQR lives in the protocol interface, not in the oracles, but the oracles reach it in four ways:

- $\mathsf{startDR}$ derives SPQR's initial secret from the handshake key, along with the Double Ratchet's;
- adversarially chosen send coins are also SPQR's coins, including those for ML-KEM key generation and encapsulation;
- corrupting a party leaks its SPQR state with the rest of its ratchet state;
- injected messages may carry arbitrary SPQR headers.

Handshakes. Both H_{lib} and H_{ro} are one-key handshakes with Kyber1024 (round 3) as the KEM. Prior analyses [52, 28] give each party two independent identity keys, one for Diffie-Hellman and one for signing. Standalone Lean proofs for that two-key model, and for its first-message authentication game, are preserved outside the trusted base.

Budgets. As in [52], the handshake budgets are per party. Each of at most N_{id} honest parties publishes a bounded number of signed-prekey bundles, each bundle a bounded number of one-time prekeys and of sessions accepted with and without one, and each party initiates a bounded number of sessions with and without a one-time prekey. We write N_{spk}, N_{opk}, N_{send} and N_{acc} for the resulting totals of signed prekeys, one-time prekeys, initiated sessions and accepted sessions, $N_{\text{send,opk}} \le N_{\text{send}}$ for the initiated sessions that use a one-time prekey, and $N_{\text{dh}} = N_{\text{id}} + N_{\text{spk}} + N_{\text{opk}} + N_{\text{send}}$ for the number of X25519 key pairs that the challenger generates. The adversary makes at most q_{hs} handshake queries in total, of which at most q_{on} are not random-oracle queries (with separate caps on random-oracle queries), and q_{start} promotions. The k-th promoted session allows at most q_k ratchet queries, where $2(q_k + 1) + 25002 < 2^{32}$, and q_{ep} epochs (ACD19's epoch cap).

C.2. Reductions and Their Costs

A reduction R runs the adversary A exactly once, as a black box and without rewinding. It prepares A's input, answers each of A's oracle queries by running a handler, and computes its own output from A's. A handler may compute on ordinary data (numbers, lists, tables), sample uniform coins and query the oracles of the assumption game. It operates on keys and group elements only through declared primitive operations, and it loops only up to a bound written in the program. If the assumption game has a challenge phase, R may also pause A at one query, send a value to the game, and resume the same run of A with the game's answer.

Costs are computed from the program text alone. Primitive operations, samples and oracle queries are charged from a fixed table per primitive. The *cost* of R against an adversary that makes q game queries is

$$\text{cost}(R, q) \;=\; c_{\text{init}} + q \cdot \max(c_{\text{coin}},\, c_{\text{handle}} + c_{\text{codec}}) + c_{\text{finish}},$$

the syntactic costs of setup, of forwarding a coin or answering one query (including its encoding), and of the output step. For reductions that pause A, the handlers before and after the pause, and the pause step itself, enter through maxima. The cost bounds R's work outside A, and a similar count, made syntactically or stated over the run, bounds R's queries to the assumption oracles. R is *cost-bounded* if three conditions hold. First, its cost and its query count are bounded by explicit polynomials in the game's budgets, independent of A. Second, no primitive operation or sample produces values larger than its declared cost allows, so that unit costs cannot hide large data; hence, on every run in which A's queries and the oracle answers have bounded size, all values that R computes have bounded size too. Third, its program has at most $2^{24} + 2^{12}(\lfloor \log_2 b \rfloor + 1)$ nodes, where b is its cost budget, so that R cannot carry a large table, such as a table of discrete logarithms. A few fixed wrappers in our statements lie outside this language (§C.6). They are part of the trusted base.

C.3. Lemmas

Lemma 1 (PQXDH)**.** *For all primitives that satisfy the well-formedness conditions of Appendix C.7, functional correctness holds. Moreover, for all handshake budgets (numbers of parties, prekeys and sessions), there are reductions, indexed by the adversary's query budget, R_{Sig}, R_{ShCr} and R_{GapDH}, against EUF-CMA of XEdDSA in the presence of the X25519 decisional DH oracle, SH-CR of Kyber1024 [28],*

and single-pair Gap-DH on X25519 (modulo the cofactor) with an XEdDSA signing oracle on the first challenge key (queried at most $2N_{\mathrm{spk}}$ times) respectively, such that:

1. *each reduction is cost-bounded; and*
2. *for all primitives that satisfy the well-formedness conditions of Appendix C.7 and for which Kyber1024 fails to decapsulate with probability at most δ_{kem}, and for every adversary B of the key-indistinguishability game of [52] with H_{ro} that respects the budgets and makes at most q_{on} handshake queries other than random-oracle queries,*

$$\mathsf{KI}_{H_{\mathrm{ro}}}(B) \;\le\; \beta^{\delta}_{\mathsf{KI}} \;:=\; \beta_{\mathsf{KI}} + 2\,N_{\mathrm{send}} N_{\mathrm{spk}}\,\delta_{\mathrm{kem}},$$

where ε_X is the success probability of R_X run with B in the game of assumption X (forging, finding a collision, solving Gap-DH) and

$$\beta_{\mathsf{KI}} = n_{\mathrm{id}}\,\varepsilon_{\mathrm{Sig}} + \tfrac{n_{\mathrm{dh}}^2}{2^{250}} + n_{\mathrm{spk}} n_{\mathrm{send}}\,\varepsilon_{\mathrm{ShCr}} + \tfrac{n_{\mathrm{spk}}(n_{\mathrm{spk}}-1)}{2^{256}} + 2\big(n_{\mathrm{spk}}(n_{\mathrm{id}}+n_{\mathrm{send}}) + n_{\mathrm{id}} n_{\mathrm{send}} + n_{\mathrm{opk}} n_{\mathrm{send,opk}}\big)\,\varepsilon_{\mathrm{GapDH}},$$

with $n_X = \min(q_{\mathrm{on}}, N_X)$ for each subscript X and $N_{\mathrm{dh}} = N_{\mathrm{id}} + N_{\mathrm{spk}} + N_{\mathrm{opk}} + N_{\mathrm{send}}$. The fractions bound collisions of public keys. Since $n_X \le N_X$, β_{KI} is at most the same expression with the totals. In Theorem 2, $B = R^{A'}_{\mathrm{KI}}$, whose online-query budget is $q_{\mathrm{on}} + 2\min(q_{\mathrm{start}}, N_{\mathrm{send}})$ by Lemma 3.

Here δ_{kem} bounds the probability of a decapsulation failure, averaged over honest key-generation and encapsulation coins. For Kyber1024, the specification gives $\delta_{\mathrm{kem}} \le 2^{-174}$, a heuristic estimate that we take as a hypothesis, so the last term is negligible.

Lemma 2 concerns one ratchet session in isolation. Let $\mathrm{SM}(id_A, id_B, s)$ be the ACD19 secure-messaging game with epoch cap q_{ep} (and ACD19's delay parameter $\Delta = 3$) on a single Triple Ratchet session between identity keys id_A, id_B, whose initial keys are uniform and whose responder's signed-prekey secret is derived from coins s and leaked. It has the corruption and injection restrictions of §C.6 built in. In the composition, the rest of the game determines a distribution O, the *prior*, over tuples $o = (id^o_A, id^o_B, s^o, A_o)$ of a context and an adversary A_o that attacks the session in that context. Its advantage is the bias

$$\mathsf{Adv}^{\mathrm{SM}}(O) \;=\; \big|\Pr[\mathrm{SM}(id^o_A, id^o_B, s^o; A_o) = 1] - \Pr[\,\cdot\, = 0]\big|, \qquad o \leftarrow O,$$

where o is drawn once before the game runs. Lemma 2 is stated for priors rather than fixed contexts so that most assumptions are charged the advantage of a single reduction.[2]

Lemma 2 (Single session)**.** *There is a map D from contexts to reductions, fixed before the primitives (each member is one program, run with whatever primitives are supplied), such that:*

1. *each reduction in D is cost-bounded: against adversaries making at most q queries its cost is at most $2^{12}(q'+1)^2(25001 + \lfloor \log_2(q_{\mathrm{ep}}+1)\rfloor)$, where $q' = q$, or $q' = \min(2q, q+2)$ for the FS-AEAD reductions of the terms for Figure 8 of ACD19; and*
2. *for all primitives satisfying P1–P10 (Appendix C.8), all q with $2(q+1) + 25002 < 2^{32}$, all δ bounding the probability that $\mathsf{dh}(x, \mathsf{base})$ equals a given point for uniform x, and every prior O such that every A_o in its support makes at most q queries,*

$$\mathsf{Adv}^{\mathrm{SM}}(O) \;\le\; \mathsf{Sess}(O) + 2\,\varepsilon^{\mathrm{SM}}_{\mathrm{mlkem}}(O),$$

where $\varepsilon^{\mathrm{SM}}_{\mathrm{mlkem}}(O)$ is the probability that, in the same game with the same adversaries, the coins of one or two sends give a failing ML-KEM-768 exchange.

[2] For most terms, Sess charges the advantage of one reduction that samples o and runs the per-o reduction R_o. With δ_o the signed advantage of R_o, this is $|\mathbb{E}_o \delta_o|$. Per-o bounds averaged inside the composition give only $\mathbb{E}_o|\delta_o| \ge |\mathbb{E}_o \delta_o|$, which is not the advantage of any single reduction. The hidden-salt terms of Sess are of this weaker, averaged form (Appendix C.9).

The bound $\mathsf{Sess}(O)$, given in full in Appendix C.9, has three kinds of terms. Some follow ACD19's analysis of the Double Ratchet core, with CKA, PRF-PRNG and FS-AEAD terms. Some cover SPQR and the leaked signed prekey, with HKDF PRF and PRG terms. The rest are statistical terms that do not depend on O. Each computational term is one of three kinds: the advantage of an adversary that samples $o \leftarrow O$ and runs A_o through a member of D (in some cases behind a simulator fixed in the statement), the advantage of such an adversary run through a fixed program, or, for the hidden-salt terms, the average over o of the advantage of A_o run through a member of D. Only the members of D are cost-bounded in the lemma. The fixed programs are explicit; apart from a few wrappers (§C.6), they are written in the costed language.

Lemma 3 reduces the security of the composed game to the key indistinguishability of the handshake and to single-session games, one per promoted session, against priors that the reduction constructs. It holds for any handshake H that implements the oracles of the key-indistinguishability game of [52] and satisfies a lazy-sampling condition, defined below. We write $\mathsf{KI}_H(B)$ for the advantage of B in that game run with H's oracles.

Lemma 3 (Composition)**.** *For all handshake budgets, every handshake H, every choice of the other primitives and every epoch cap q_{ep}, there are a reduction R_{KI} against key indistinguishability of H and, for each k, a reduction R_k^{pre} against the SM game that pauses A at one query, such that for all query budgets q_{hs}, q_{on}, q_{start} and $(q_k)_k$, with $q_{\mathrm{tot}} = q_{\mathrm{hs}} + q_{\mathrm{start}} + \sum_{k<q_{\mathrm{start}}} q_k$, $n_X = \min(q_{\mathrm{hs}}, N_X)$ and $n_{\mathrm{inst}} = \min(q_{\mathrm{start}}, n_{\mathrm{send}})$:*

1. *each reduction is cost-bounded: against adversaries making at most q_{tot} queries it costs at most $2^{10}(q_{\mathrm{tot}}+1)\big(q_{\mathrm{hs}} + n_{\mathrm{opk}} + n_{\mathrm{inst}} + \max_k q_k + \lfloor \log_2 q_{\mathrm{ep}} \rfloor + 10(n_{\mathrm{id}} + n_{\mathrm{spk}} + n_{\mathrm{send}} + n_{\mathrm{acc}}) + \ell + 25001\big) + 2^{13} n_{\mathrm{inst}}(n_{\mathrm{id}} + n_{\mathrm{spk}} + n_{\mathrm{acc}} + 1)$; R_{KI} makes at most $2q_{\mathrm{tot}}$ queries, and R_k^{pre} makes none before its pause and at most q_{tot} in total; and*
2. *for every adversary A of $G(\Sigma^H_{\mathrm{model}})$ that respects the budgets:*
   (a) *R^A_{KI} respects the budgets of [52], with q_{hs} and q_{on} each increased by $2\min(q_{\mathrm{start}}, N_{\mathrm{send}})$;*
   (b) *for each $k < q_{\mathrm{start}}$ and bit f, running R_k^{pre} with A up to its pause (which does not depend on f) and then resuming it with bit f yields a prior $O_k(f)$ in whose support every adversary makes at most q_k queries; and*
   (c) *if H satisfies the lazy-sampling condition for A, then*

$$\mathsf{Adv}^{\mathrm{comp}}(A) \;\le\; 2\,\mathsf{KI}_H(R^A_{\mathrm{KI}}) \;+\; \sum_{k<q_{\mathrm{start}}} \max_{f\in\{0,1\}} \mathsf{Adv}^{\mathrm{SM}}\big(O_k(f)\big).$$

The *lazy-sampling condition* concerns H together with the game's driver. Fix $k < q_{\mathrm{start}}$ and a bit b_k, and consider the hybrid in which ratchet instances $i < k$ use challenge bit 1, instance k uses bit b_k and instances $i > k$ use bit 0, run against H in the random-key world. The condition requires that this hybrid has the same output distribution whether instance k is keyed with the key derived from H's test answer or with a fresh uniform key. A condition on H alone ("a fresh key in place of one tested key is invisible") would fail for every partner-consistent handshake, since the adversary could test both partners. The driver withholds the test query and promotes at most one instance per ephemeral key. We prove the condition for H_{lib} for every adversary.

Lemma 3 does not use P1–P10. Its per-session terms are raw single-session biases, to which Lemma 2 is then applied with prior $O_k(f)$. We write $\mathsf{Sess}_k = \max_f \mathsf{Sess}(O_k(f))$.

Lemma 4 concerns the machine code. Let enc encode a model state as a party's directory (the files of Figure 4, with each session stored as `libsignal`'s serialized record), and let obs map a model outcome to an exit code, a standard output and new file contents. A party state σ is *well-formed* if its session, if any, is one reachable from a PQXDH handshake followed by ratchet steps, every file of $\mathsf{enc}(\sigma)$ is below 2^{31} bytes, and its prekey identifiers of each kind are distinct. For an operation op

(`init` with no session, `send`, or `recv` of a prekey message in protobuf's canonical encoding, of an ordinary message, or of anything else), step is its model function and run runs the executable from the state in which Linux has loaded it to its `exit_group` system call.

Lemma 4 (Machine code refines the model)**.** *Let the executable be the bytes of our `libsignal-cli` build. For every placement of the library and of musl's allocator that matches these bytes, all hooked primitives (the hooked routines of §4 return their values, and* step *uses the same ones), every operation* op*, every well-formed σ, every standard input x below 2^{32} bytes (for prekey messages, inner message plus 2048 bytes below 2^{32}) and every `coins` file r:*

1. *either* $\mathsf{run}(\mathsf{op}, \mathsf{enc}(\sigma), x, r) = \mathsf{obs}(\mathsf{step}(\sigma, x; r))$*, an equality of outcomes including the error cases, or the run exhausts the machine's memory (we claim nothing about its output or files; §4.1 shows that this does not happen);*
2. *exception: if x is a prekey message that does not match the session the party holds, clause 1 holds only for the exit code, the standard output and the files other than the session file, and with* step *taken from the same party with no session.*

"No terminal state" is an outcome that the model never produces, so clause 1 includes termination. A rejected operation exits with code 1 and writes no file. The exception arises because, when it accepts such a message, `libsignal` archives the old session in the new record, which our encoding does not represent. The game never reaches the accepted case: the initiator refuses every prekey message, and the responder accepts one over an existing session only if its header is that of the session's handshake. The lemma assumes nothing about the hooked primitives. That the real routines behave as their Lean specifications is part of the trusted base (§5.3). The lemma is functional refinement of whole runs, and intermediate machine states are unconstrained. For SPQR, step uses our Lean model of `spqr` 1.6.0, which is trusted. The lemma fixes every output, including on rejected and adversarial inputs, so any deviation from the model, harmful or not, contradicts it, whether it comes from the Rust source or the compiler.

Proof outline. The proof is compositional, in a separation logic for AArch64. Each compiled function, including inlined Rust library code (`Vec`, protobuf, `SessionRecord` serialization, SPQR), the CLI's argument handling and file-backed stores, and musl's start-up and exit code and its `mallocng` allocator, gets a contract relating its registers and memory footprint to the corresponding model function. Contracts are proved by symbolic execution with a weakest-precondition calculus (callee contracts, hook specifications, loop invariants, the frame rule, and bit-vector side conditions proved in Lean's kernel). Since the semantics is deterministic, the contract of the whole run determines its unique terminal state. □

C.4. The Main Theorem in Full

Let Σ_{bin} be the interface in which every ratchet step is a run of `libsignal-cli` in the party's directory, with messages passed on standard input and output. On startDR, the initiator runs `init`, and the responder's first `recv` processes her prekey message. The challenger writes no session file, so every session record a party holds was written by its own run of the executable. The handshake is `libsignal`'s, H_{lib}, and $\varepsilon_{\text{mlkem}}(A)$ is as in §5.2.3. The following is the full statement of Theorem 2.

Theorem 2 (Binary security, full statement)**.** *For every placement of the library and of musl's allocator that matches the bytes of the executable, all handshake budgets and every fixed message length $\ell < 10^6$ (the plaintext length of every ratchet message), there are an* adapter*, a reduction that turns an adversary of $G(\Sigma_{\text{bin}})$ into one of $G(\Sigma_{\text{model}}^{H_{\text{lib}}})$, and reductions, all cost-bounded for all query budgets, such that for*

all hooked primitives whose derived handshake primitives satisfy the well-formedness conditions of Appendix C.7 and decapsulate correctly with probability at least $1 - \delta_{\mathrm{kem}}$*, and that satisfy P1–P10 (Appendix C.8), for all* δ *as in Lemma 2, all query budgets with* $2(q_k + 1) + 25002 < 2^{32}$ *for every session, and every adversary* A *of* $G(\Sigma_{\mathrm{bin}})$ *respecting the budgets:*

1. *Security.*
$$\mathsf{Adv}^{\mathrm{bin}}(A) \;\le\; 2\,\varepsilon_{\mathsf{KI}} + 2\,N_{\mathrm{send}} N_{\mathrm{spk}}\,\delta_{\mathrm{kem}} + \textstyle\sum_{k<q_{\mathrm{start}}} \mathsf{Sess}_k + 2\,\varepsilon_{\mathrm{mlkem}}(A),$$
where $\varepsilon_{\mathsf{KI}} = \mathsf{KI}_{H_{\mathrm{lib}}}(R^{A'}_{\mathrm{KI}})$ *and* Sess_k *are as in Lemmas 3 and 2 for the adversary* A' *that runs* A *through the adapter.*
2. *If moreover* $\mathsf{KI}_{H_{\mathrm{lib}}}(R^{A'}_{\mathrm{KI}}) \le \mathsf{KI}_{H_{\mathrm{ro}}}(R^{A'}_{\mathrm{KI}}) + \varepsilon_{\mathrm{ro}}$ *(the random-oracle idealization, Appendix C.8), then the bound of clause 1 holds with* $2\,\varepsilon_{\mathsf{KI}}$ *replaced by* $2(\beta^{\delta}_{\mathsf{KI}} + \varepsilon_{\mathrm{ro}})$*, where* $\beta^{\delta}_{\mathsf{KI}}$ *is the bound of Lemma 1 for* $B = R^{A'}_{\mathrm{KI}}$.

The bound is that of Lemma 3, with each $\mathsf{Adv}^{\mathrm{SM}}$ term replaced by Sess, plus the decapsulation term, and with the per-session ML-KEM-768 terms of Lemma 2 replaced by the single term $2\,\varepsilon_{\mathrm{mlkem}}(A)$. In clause 2, the Kyber1024 decapsulation failure probability enters twice, inside $\beta^{\delta}_{\mathsf{KI}}$ and as the separate term. These are different bad events, one in the key-indistinguishability game and one in the coupling below. Every reachable session record encodes to at most 1,209,331 bytes, whatever q_k is, and every honest input is below 2^{32} bytes, so the size premises of Lemma 4 hold on every run. Injected inputs must be below 2^{32} bytes, and the inner message of an injected prekey message below $2^{32} - 2048$ bytes, leaving room for the wrapper. The game refuses other inputs without running the executable. The reductions of Lemmas 1 and 2 are fixed before the hooked primitives. The adapter depends only on the public-key encoding, and the composition reductions of Lemma 3 depend on all primitives, because they run the code of the other sessions.

Proof. The Lean proof combines Lemma 1, Lemma 4 and the no-out-of-memory theorem (§4.1), used through their statements, with a composition argument that follows Lemmas 3 and 2 but is carried out directly, in a game that stops at the first ML-KEM-768 decryption failure. This is needed because the ML-KEM-768 term of Lemma 2 is evaluated on adversaries built by the composition reductions, which can trigger a failure with probability 1; the halted game charges the failure once, on A itself.

(i) *Correctness of every run.* By Lemma 4, every run in the game (`init`, the first prekey `recv`, and every later `send` and `recv`) has the model's outcome unless the machine runs out of memory, which the no-out-of-memory theorem rules out. This includes the compiled `spqr` state machine, which computes what our Lean model of SPQR predicts. Only its ML-KEM-768 and HMAC routines are hooked (§4).

(ii) *Coupling.* The binary game stores directories, and the model game stores model states. The coupling invariant is that every directory is $\mathsf{enc}(\sigma)$ for the corresponding reachable σ, and (i) preserves it at every oracle call. It breaks only if a Kyber1024 decapsulation in an honest handshake fails, which costs at most $2\,N_{\mathrm{send}} N_{\mathrm{spk}}\,\delta_{\mathrm{kem}}$. Otherwise A in $G(\Sigma_{\mathrm{bin}})$ and A' in $G(\Sigma^{H_{\mathrm{lib}}}_{\mathrm{model}})$ win with the same probability, and A' respects the same budgets.

(iii) *Security of the model.* Bound the advantage of A' by the hybrid argument of Lemma 3 with $H = H_{\mathrm{lib}}$, whose lazy-sampling condition holds for every adversary, and each per-session term as in Lemma 2 with $q = q_k$, in the game halted at the first ML-KEM-768 failure; reaching that failure costs at most $2\,\varepsilon_{\mathrm{mlkem}}(A)$. Clause 2 follows by applying Lemma 1 to $R^{A'}_{\mathrm{KI}}$. □

C.5. Assumptions in Detail

Joint security. Because `libsignal` uses a single Curve25519 scalar per party for both X25519 Diffie-Hellman and XEdDSA bundle signatures [79], Lemma 1 pairs its signature and Diffie-Hellman games following standard joint-security definitions [46, 95]. XEdDSA EUF-CMA gives the adversary the X25519 decisional DH oracle (`GapSigEufCma`), and single-pair X25519 Gap-DH gives the adversary an XEdDSA signing oracle on the first challenge key (`GapDhWithSign`).

Outside the trusted base, we mechanize SHA-512 random-oracle reductions from both joint assumptions to standalone X25519 Gap-DH for `libsignal`'s randomized XEdDSA [95, 35, 97] under an abstract prime-order group-algebra witness (`XeddsaCurveLaws`, Appendix C.10). These are a single-run reduction for `GapDhWithSignRo` with a factor-2 loss from guessing the Edwards sign bit, a two-run DDH-checked variant with factor 1, and a rewinding Forking-Lemma reduction for `GapSigEufCmaRo` with a $\sqrt{2(q_H+1)\,\mathrm{Adv}^{\mathrm{GapDH}}(B_2)}$ bound. Lemma 1 and Theorem 2 retain `GapDhWithSign` and `GapSigEufCma` directly, for three reasons: β_{KI} avoids the square-root degradation of the Forking Lemma across N_{id} identities, each reduction in the theorem remains a single-run program in the costed language without rewinding, and the theorem avoids assuming a SHA-512 random oracle or concrete Curve25519/Ed25519 field arithmetic.

Environment. The statement assumes functional properties of the hooked ML-KEM-768, AES-256-CBC and X25519 routines, such as that decryption inverts encryption (Appendices C.7 and C.8). Everything else about the environment is part of the trusted base (§5.3):

- the hooked routines of §4 (low-level cryptography, with AES-256-CBC including padding hooked as one function; five `libc` memory and string routines such as `bcmp`; the two functions that write files and standard output; and a few small Rust library routines) behave as their Lean specifications say, write only their declared outputs (no other caller-saved register), and fail only as specified (e.g., no full disk);
- the clock reads the UNIX epoch;
- each call is a fresh, non-setuid process at a fixed load address (no ASLR), with only standard input, output and error open, an empty environment, an 8 MiB stack, 1 GiB of available memory and fixed kernel random bytes (§4.1); and
- `spqr`'s CPU-feature probe reports no AES or PMULL instructions, so the proofs follow the portable code paths.

C.6. Restrictions in Detail

Adversary and game. Besides the restrictions in §5.2.5, receive coins are honest, the responder does not send first in the first epoch, and injected ratchet keys must lie in the prime-order group. A few adversarial actions (revealing any signed-prekey secret, or corrupting the initiator's identity key of any session before its responder accepts) make all later ratchet queries, in every session, return a default answer. Wins recorded before such an action still count, and the handshake test remains available. However, the theorem does not make claims about attacks that combine such an action with queries to an unrelated session. The handshake game is the variant of [52] with registered secret keys, no randomness reveals and a peer that is uncorrupted at session start. Smaller restrictions are listed in the artifact's deviation log.

Setting. Budgets are fixed per instance. There is no self-session, and Bob's Kyber prekey is always a last-resort key. With 8-byte MACs, the statistical terms are multiples of $\mu \geq 2^{-64}$ that grow

polynomially in q_k and q_{ep} (quadratically in each, per session, and summed over sessions; see §5.2.5 and Appendix C.9).

Binary. Prekey messages are covered only in canonical protobuf encoding. The requirement of the promotion oracle that the bundle match published keys is argued, not mechanized, to lose no generality. The verified executable, on which both the computational and the symbolic theorems are stated, is built with `rustc`'s release profile with one codegen unit and `panic=abort`, linked at a fixed base address, and has each hooked routine linked as a stub of four zero bytes, which traps if executed. Each run starts with SPQR's CPU-feature probe already set to report no AES or PMULL support, so SPQR takes its software code paths.

Cost model. The wrapper that refuses the corruptions the game forbids, and the glue that runs the reductions on the adversary, are plain Lean functions rather than programs in the reduction language; they are part of the trusted base. All other reductions in the statement, including the simulators for Figure 8 of ACD19, are cost-bounded.

C.7. Well-Formedness of the Handshake Primitives

Lemma 1 quantifies over primitives that satisfy the following conditions. They quantify over all inputs, in some cases restricted to canonical, on-curve or correctly sized ones as stated; in particular the adversary may supply any 32 bytes as an X25519 public key, including small-order and twist points. That the real primitives satisfy them is a trusted premise.

- W1 *Lengths and totality.* Every primitive returns outputs of the expected byte lengths (hashes, AES blocks, X25519 values, scalars, Edwards points, Kyber1024 keys and ciphertexts), Kyber1024 encapsulation and decapsulation succeed on well-sized inputs, and SPQR initialization always succeeds.
- W2 *Agreement with reference definitions.* SHA-512, the HKDF that derives the session key, the root and chain key derivations, message-key derivation, AES-256-CBC with HMAC, XEdDSA verification and the canonicity check on public keys agree with reference definitions written in Lean.
- W3 *Algebraic laws of X25519.* X25519 is invariant under scalar clamping and commutes on base-point products; a product of the base point never yields the all-zero output; whether an output is all-zero does not depend on the scalar; the base point is canonical and canonical inputs give canonical outputs; multiplication by the cofactor 8 commutes with X25519; the signing public key is the X25519 public key; and the Gap-DH oracle agrees with its reference predicate, is symmetric, and is consistent with X25519 in either argument.
- W4 *Cofactor projection.* The projection onto the prime-order subgroup is idempotent; X25519 outputs depend on a public key only through its projection, and conversely equal outputs under one scalar imply equal projections; scalars with the same public key agree on every on-curve input; the projection is injective on canonical keys; and only 32-byte strings project onto the projection of a public key.
- W5 *Correctness.* AES-256 block decryption inverts encryption, the AEAD decrypts what it encrypts, XEdDSA signatures verify, and SPQR's initial send state has a matching receive state with the same salt; every salt SPQR's receive initialization produces, on any input, has at most 32 bytes.
- W6 *Key collisions.* A fresh X25519 public key has the same projection as a given one with probability at most 2^{-250}, and a fresh Kyber1024 public key equals a given one with probability at most 2^{-255}. (The bound 2^{-250} rather than 2^{-251} is tight for clamped scalars. The bound 2^{-255} rather

than 2^{-256} is needed because the Kyber1024 public key depends only on the 32-byte seed d, through G = SHA3-512. If G is a random function, some two of the 2^{256} seeds collide with probability $1 - e^{-1/2} \approx 0.39$, and then the bound 2^{-256} per target key, a property of the fixed function G, fails; three seeds collide only with probability about $2^{-258.6}$, so the bound 2^{-255} holds almost surely.) These give the two collision terms of β_{KI}.

Kyber1024's decapsulation correctness is not among these conditions; it enters Lemma 1 separately, as the bound δ_{kem}.

C.8. Primitive and X25519 Premises

The Diffie-Hellman group is abstract: a type Pt with a zero element, a base point base, a function $\mathsf{dh} : \{0,1\}^{256} \times \mathsf{Pt} \to \mathsf{Pt}$, an encoding $\mathsf{enc} : \mathsf{Pt} \to \{0,1\}^{256}$ used inside the KDFs, and the encoding encT/decoding decT of public keys in `libsignal`'s wire format. Lemma 2 and Theorem 2 assume the following (P1–P3 concern ML-KEM-768, AES and the message length; P4–P10 concern X25519).

- P1 ML-KEM-768's first encapsulation step succeeds on every 64-byte header (`Encaps1Total`; `libcrux` checks only the length).
- P2 AES-256-CBC decryption inverts encryption: $\mathsf{D}_k(\mathsf{E}_k(x)) = x$.
- P3 The message length satisfies $\ell < 10^6$.
- P4 dh commutes: $\mathsf{dh}(a, \mathsf{dh}(b, \mathsf{base})) = \mathsf{dh}(b, \mathsf{dh}(a, \mathsf{base}))$.
- P5 dh is the X25519 function of the model (`LibX25519Rel`): enc is injective with 32-byte values, $\mathsf{enc}(0) = 0^{256}$, enc(base) is the X25519 base point, base $\neq 0$; the model's byte-level $\mathsf{x25519}(x, \mathsf{enc}(P)) = \mathsf{enc}(\mathsf{dh}(x, P))$, and likewise for public-key derivation; dh is invariant under clamping of x; $P \neq 0$ implies $\mathsf{dh}(x, P) \neq 0$; and the bytes of $\mathsf{encT}(P)$ are $\mathsf{enc}(P)$.
- P6 $\mathsf{decT}(\mathsf{encT}(P)) = P$.
- P7 encT is injective (immediate from P6, so not a separate hypothesis in Lean).
- P8 $\mathsf{encT}(P)$ is never the dummy key $(\texttt{0xff})^{32}$.
- P9 $\mathsf{dh}(x, 0) = 0$.
- P10 All scalars have the same kernel on Pt: $\mathsf{dh}(x, Q) = \mathsf{dh}(x, Q') \iff \mathsf{dh}(s, Q) = \mathsf{dh}(s, Q')$ for all s, x, Q, Q' (`DhKerUniform`).

P10 holds for clamped X25519 on the prime-order subgroup (clamping excludes $x \equiv 0$ modulo the group order, leaving at most two clamped scalars per x-only point and giving $\delta = 2^{-250}$). Including nonzero torsion points ($E[8] \setminus \{0\}$ or twist $E'[4] \setminus \{0\}$) in Pt would violate P5's $\mathsf{dh}(x, P) \neq 0$ under clamped scalars $c = 8m$, and including mixed-order or twist cosets would make CKA trivially distinguishable (with advantage at least 7/8) because honest public-key generation lands in the prime-order subgroup while challenges sample uniformly from Pt; thus Pt must be the prime-order subgroup. As a sanity check, the Lean development proves that P1–P10 and W1–W6, together with $\delta = 2^{-250}$ and $\delta_{\mathrm{kem}} = 0$, are jointly satisfiable on a non-trivial x-only group of prime order $\ell = 2^{252} + 27742317777372353535851937790883648493$ ($|\mathsf{Pt}| = (\ell + 1)/2 > 2^{251}$).

Random-oracle idealization (not a premise). One further hypothesis enters only clause 2 of Theorem 2, as a hypothesis on the particular adversary $R_{\mathrm{KI}}^{A'}$ for a given $\varepsilon_{\mathrm{ro}}$; clause 1 does not use it.

- G1 $\mathsf{KI}_{H_{\mathrm{lib}}} \leq \mathsf{KI}_{H_{\mathrm{ro}}} + \varepsilon_{\mathrm{ro}}$ (`KeyIndRoGap`): the random-oracle step, i.e. the named ROM instantiation `HkdfRO` (DEV-PQ14) of the session-key HKDF on the one-key game. In the random-oracle model HKDF is replaced by `HkdfRO`, the two games coincide, and one takes $\varepsilon_{\mathrm{ro}} = 0$; for the concrete HKDF this is the standard ROM heuristic (the same one Lemma 1 rests on), not a theorem.

C.9. The Terms of Sess

Fix a prior O and let R_o be the member of D for the context of o. Besides O, Sess depends on q, q_{ep}, δ and a fixed point used by the simulators. For an assumption X and a reduction B_o chosen per leaf, $\varepsilon^X(\overline{B})$ is the advantage against X of the averaged adversary that samples $o \leftarrow O$ and runs B_o (on A_o, through the simulators of R_o), and $\mathbb{E}\,\varepsilon^X(B)$ is $\mathbb{E}_{o\leftarrow O}\,\varepsilon^X(B_o)$. All adversaries in U below run A_o inside the wrapper that refuses corr-B while B is in epoch 0 or A is in an epoch below 10. With $T(t) = \{-\infty\} \cup \{0, \ldots, t-3\}$ the guesses of the leak epoch, $T_2(t) = \{-\infty\} \cup \{0, \ldots, t-2\}$, and $m(q) = 2q + 25001$ the number of forward-secure slots and $q' = \min(2q, q+2)$,

$$\begin{aligned}
\mathsf{Sess}(O) &= U(O) + 2\,\varepsilon^{\mathrm{PRF}}_{\mathrm{ext}}(\overline{R}^{\mathrm{hop1}}) + 2\,\varepsilon^{\mathrm{PRF}}_{\mathrm{ext}}(\overline{R}^{\mathrm{hop3}}) \\
&\quad + 2\Big(\gamma_{\mathrm{forge}} + \textstyle\sum_{j<9}\big(\varepsilon^{\mathrm{PRG}}_{\mathrm{exp}}(\overline{R}^{\mathrm{hop2}}_{\mathrm{prg},j}) + \varepsilon^{\mathrm{PRF}}_{\mathrm{ext}}(\overline{R}^{\mathrm{hop2}}_{\mathrm{prf},j})\big) \\
&\quad + \gamma_{\mathrm{create}} + \gamma_{\mathrm{fresh}} + \gamma_{\mathrm{coll}} + \gamma_{\mathrm{guess}}\Big), \\
U(O) &= 2\,\mathrm{F8A} + \mathrm{F8P} + 2\rho + 2\big(\varepsilon^{\mathrm{PRG}}_{\mathrm{off}}(\overline{E}_0) + q\,\mu_0\big) + 2\,\mathrm{HS} + 2\,\varepsilon^{\mathrm{PRG}}_{\mathrm{off}}(\overline{E}_{\mathrm{u}}),
\end{aligned}$$

where $\overline{E}_0$ and $\overline{E}_{\mathrm{u}}$ are the epoch-0 and uniform-start distinguishers of R_o, averaged over o, and

$$\begin{aligned}
\mathrm{F8A} &= 2\sum_{t<q_{\mathrm{ep}}+3}\ \sum_{t_L\in T(t)}\Big(\varepsilon^{\mathrm{CKA}}(\overline{R}^{\mathrm{A}}_{\mathrm{cka}}) + \varepsilon^{\mathrm{PP}}(\overline{R}^{\mathrm{A}}_{\mathrm{pp}}) + \textstyle\sum_{j<q}\mathrm{FS}(\overline{R}^{\mathrm{A}}_{\mathrm{fs},j})\Big), \\
\mathrm{F8P} &= 2\sum_{i<q_{\mathrm{ep}}+2}\ \sum_{t_L\in T(i)}\Big(2\varepsilon^{\mathrm{CKA}}(\overline{R}^{\mathrm{P}}_{\mathrm{cka}}) + 2\varepsilon^{\mathrm{PP}}(\overline{R}^{\mathrm{P}}_{\mathrm{pp}}) + \mathrm{FS}(\overline{R}^{\mathrm{P}}_{\mathrm{fs}}) + 2\,\mathrm{FS}(\overline{R}^{\mathrm{P}}_{\mathrm{coin}})\Big), \\
\mathrm{FS}(\overline{B}) &= 2\sum_{j<m(q')}\Big(\varepsilon^{\mathrm{PRG}}_{\mathrm{chain}}(\overline{B}_j) + 2\,\varepsilon^{\mathrm{PRF}}_{\mathrm{salt}}(\overline{B}_j) + \varepsilon^{\text{OT-CCA}}_{\mathrm{wire}}(\overline{B}_j)\Big) + 4q\,\mu, \\
\rho &= \sigma_{\mathrm{Pt}} + \sigma + (q + q_{\mathrm{ep}} + 2)^2\,\delta, \\
\mathrm{HS} &= \sum_{t<q_{\mathrm{ep}}+3}\ \sum_{t_L\in T_2(t)}\Big(\mathbb{E}\,\varepsilon^{\mathrm{CKA}}(R^{\mathrm{H}}_{\mathrm{cka}}) + \mathbb{E}\,\varepsilon^{\mathrm{PP}}(R^{\mathrm{H}}_{\mathrm{pp}}) \\
&\qquad + \textstyle\sum_{j<q}\mathbb{E}\,\mathrm{FS}_{\mathrm{DF}}(R^{\mathrm{H}}_{\mathrm{fs},j}) + 4q^2\mu\Big) \\
&\quad + \sigma + (q + q_{\mathrm{ep}} + 2)^2\,\delta, \\
\mathrm{FS}_{\mathrm{DF}}(B) &= 2\sum_{j<m(q')}\Big(\varepsilon^{\mathrm{PRG}}_{\mathrm{chain}}(B_j) + 2\,\varepsilon^{\mathrm{PRF}}_{\mathrm{salt}}(B_j) + \varepsilon^{\text{OT-CCA}}_{\mathrm{wire}}(B_j)\Big).
\end{aligned}$$

Here CKA is the DH-based CKA of [7] over dh (healing after one epoch); PP is PRF-PRNG security of the root KDF; $\mathrm{PRG}_{\mathrm{off}}$ is the root KDF on a fixed input as a PRG in the root key; $\mathrm{PRG}_{\mathrm{chain}}$, $\mathrm{PRF}_{\mathrm{salt}}$ and $\text{OT-CCA}_{\mathrm{wire}}$ are the chain-key PRG, the HKDF salt PRF and the OT-CCA security of the message encryption (AES-CBC with truncated HMAC); $\mathrm{PRF}_{\mathrm{ext}}$ and $\mathrm{PRG}_{\mathrm{exp}}$ are HKDF-Extract and HKDF-Expand of the root key derivation. The reductions $R^{\mathrm{A}}, R^{\mathrm{P}}$ (Figure 8 authenticity and privacy), R^{H} (hidden salt) and $R^{\mathrm{hop}\cdot}$ are members of D; in HS they run behind two further members of D, the deliver-forge simulator and a challenge simulator. The statistical terms are suprema over fixed tests, not over adversaries:

- μ: the probability that a fixed wire verifies under a uniform message key (8-byte HMAC tag), maximized over identities, wires shorter than 2^{32} bytes and tags; $\mu \geq 2^{-64}$.
- μ_0: the same for a uniform chain key after c steps, maximized over $c < 2^{32}$, salt, identities and the wire; $\gamma_{\mathrm{forge}} = q\,\mu_0$.
- σ (σ_{Pt}): $q(q_{\mathrm{ep}} + 1)$ times the supremum, over keys, points and MAC tests, of the PRG advantage of the test against a spurious DH step of `libsignal` on an evicted epoch's key (on an arbitrary non-zero point) plus the probability that the test accepts a uniform key.

- γ_{fresh}: q times the analogous supremum for HKDF-Expand.
- $\gamma_{\text{coll}} = (q+1)\max_{a,T}\Pr_r[\mathsf{dh}(a,\mathsf{dh}(r,\mathsf{base})) = \mathsf{dh}(a,T)]$, $\gamma_{\text{create}} = 48q\max_{T,\,Q\neq 0}\Pr_r[\mathsf{dh}(r,T) = Q]$, and $\gamma_{\text{guess}} = 6q\max_P \Pr_x[\mathsf{dh}(x,\mathsf{base}) = P]$, each at most about $q\cdot 2^{-244}$ for X25519.

The members of D, including the simulators and the epoch-0 and uniform-start distinguishers, satisfy the cost budget of Lemma 2.

C.10. SHA-512 Random-Oracle Reductions for XEdDSA

Outside the trusted base, we mechanize SHA-512 random-oracle reductions from `GapDhWithSign` and `GapSigEufCma` to standalone X25519 Gap-DH modulo the cofactor 8 for `libsignal`'s randomized XEdDSA signer `xeddsaSignRef` [79, 95, 35]. Let

$$\ell = 2^{252} + 27742317777372353535851937790883648493$$

be the Curve25519/Ed25519 subgroup prime. Under Appendix C.7 and an abstract prime-order group-algebra witness `XeddsaCurveLaws` pairing the Montgomery and Edwards encodings over $\mathbb{Z}/\ell\mathbb{Z}$, `XeddsaSignRoundtrip` proves sign-then-verify correctness both for concrete SHA-512 and in the lazy SHA-512 random-oracle model, and shows that instantiating signing with `xeddsaSignRef` preserves Appendix C.7.

Loss terms and reduction bounds. Let $r_{\text{wide}} = 2^{512} \bmod \ell$ and let $\delta_{\text{wide}} = r_{\text{wide}}(\ell - r_{\text{wide}})/(\ell\cdot 2^{512}) < 2^{-261}$ be the total-variation distance between a 512-bit SHA-512 digest reduced modulo ℓ and uniform $\mathbb{Z}/\ell\mathbb{Z}$. For an adversary making q_{ro} direct SHA-512 queries, q_{ver} verification queries, q_{psig} primitive signing queries, and q_S challenger signing queries, let $q_H = q_{\text{ro}} + q_{\text{ver}} + q_{\text{psig}}$ be the effective challenge-hash budget, and define the zero-knowledge signing simulation loss and challenge modular-reduction loss:

$$\Delta_{\text{sim}}(q_S,q_H) = \frac{q_S(2q_H+3q_S)}{2^{512}} + q_S\,\delta_{\text{wide}} + \frac{q_S(q_S+q_H)}{\ell},$$
$$\Delta_{\text{mod}}(q_H) = (q_H+1)\,\delta_{\text{wide}}.$$

In the SHA-512 random-oracle games `GapDhWithSignRo` and `GapSigEufCmaRo`, programming the Fiat–Shamir oracle yields two reductions from `GapDhWithSignRo` to standalone X25519 Gap-DH: a single-run reduction B_1 (`GapDhWithSignRoBound1A`, $q_{\text{ddh}}(B_1) \le q_{\text{ddh}}(A)$, $q_{\text{prim}}(B_1) \le q_{\text{prim}}(A)$) that guesses the challenge key's Edwards sign bit, and a two-run reduction B_1' (`GapDhWithSignRoBound1B`, $q_{\text{ddh}}(B_1') \le 2q_{\text{ddh}}(A)+1$, $q_{\text{prim}}(B_1') \le 2q_{\text{prim}}(A)$) that checks its first candidate via the decisional DH oracle:

$$\mathsf{Adv}^{\text{GapDH+Sign}}_{\text{RO}}(A) \le 2\,\mathsf{Adv}^{\text{GapDH}}(B_1) + \Delta_{\text{sim}}(q_S,q_H),$$
$$\mathsf{Adv}^{\text{GapDH+Sign}}_{\text{RO}}(A) \le \mathsf{Adv}^{\text{GapDH}}(B_1') + \Delta_{\text{sim}}(q_S,q_H).$$

For `GapSigEufCmaRo`, rewinding A at its target challenge query via the VCVio general Forking Lemma [97] ($q_{\text{ddh}}(B_2) \le 2q_{\text{ddh}}(A)$, $q_{\text{prim}}(B_2) \le 2q_{\text{prim}}(A)+1$), extracting the secret scalar modulo ℓ up to sign by 2-special soundness, and lifting it to a clamped 32-byte X25519 scalar establishes both the quadratic bound (`GapSigEufCmaRoBoundQuad`)

$$\varepsilon'\Big(\frac{\varepsilon'}{q_H+1} - \frac{1}{\ell}\Big) \le 2\,\mathsf{Adv}^{\text{GapDH}}(B_2), \qquad \varepsilon' = \max\big(0,\ \mathsf{Adv}^{\text{Sig+DDH}}_{\text{RO}}(A) - \Delta_{\text{sim}}(q_S,q_H) - \Delta_{\text{mod}}(q_H)\big),$$

and the closed-form bound (`GapSigEufCmaRoBound`)

$$\mathsf{Adv}^{\text{Sig+DDH}}_{\text{RO}}(A) \le \sqrt{2(q_H+1)\,\mathsf{Adv}^{\text{GapDH}}(B_2)} + \frac{q_H+1}{\ell} + \Delta_{\text{sim}}(q_S,q_H) + \Delta_{\text{mod}}(q_H).$$

A concrete group model over $\mathbb{Z}/\ell\mathbb{Z}$ witnesses joint satisfiability of Appendix C.7 and `XeddsaCurveLaws`, and known-answer tests over GF($2^{255} - 19$) with FIPS 180-4 SHA-512 check byte-exact agreement with `libsignal` v0.105.4 test vectors.

Model boundaries. Four boundaries separate these standalone reductions from Lemma 1 and Theorem 2: (a) Appendix C.7 does not pin the primitive signing function to `xeddsaSignRef`, since `libsignal-cli` only verifies received bundle signatures; (b) `XeddsaCurveLaws` is not proved for concrete GF($2^{255} - 19$) Curve25519/Ed25519 arithmetic, as `curve25519-dalek` lies outside the binary proof scope (§5.2.4); (c) the primitive bridge in the binary theorem instantiates SHA-512 as the concrete Merkle–Damgård hash function rather than a lazy random oracle; and (d) the $\sqrt{2(q_H + 1)\,\mathsf{Adv}^{\mathrm{GapDH}}(B_2)}$ Forking-Lemma loss makes the signature reduction qualitative rather than concrete-tight at deployment budgets ($N_{\mathrm{id}} = 2^{20}$, $q_H = 2^{64}$).