Mathematically Proving the Security of the Signal Messaging Protocol

Daniel Dia [1]

[1]Department of Electrical & Computer Engineering (ECE), American University of Beirut (AUB)

The Initial Handshake: X3DH

Before a conversation begins, Signal uses the **Extended Triple Diffie-Hellman (X3DH)** protocol [1] to asynchronously establish a shared secret key. This is done so that a user (Bob) can be offline while another user (Alice) sets up a secure channel. Bob uploads a set of public keys to a server:

- **Identity Key (IK):** A long-term key representing Bob's identity.
- **Signed Prekey (SPK):** A medium-term key that is signed by the IK.
- **One-Time Prekeys (OPKs):** A large batch of single-use keys.

Alice fetches this "prekey bundle", generates her own ephemeral key, and uses her own IK to perform three or four Diffie-Hellman calculations. The results are combined to create the initial shared secret, which then seeds the Double Ratchet.

The Double Ratchet: A Self-Healing Conversation

Once X3DH is complete, the conversation is protected by the Double Ratchet algorithm [2]. This provides both **Forward Secrecy** (past messages are safe if keys are stolen) and **Post-Compromise Security** (future messages are safe). It consists of two interlocking mechanisms:

- **Symmetric-key Ratchet:** For each message, a Key Derivation Function (KDF) is used to advance a "chain key." Each step produces a new message key and a new chain key [2]. This ensures one message key cannot be calculated from another.
- **Diffie-Hellman Ratchet:** Whenever a user has an opportunity to send a message after receiving one, they generate a new DH keypair and include the public key in their message. The resulting DH shared secret is used to re-seed the symmetric ratchet. This "heals" the conversation if a key was previously compromised ($\Longrightarrow$ **PCS!**) [2].

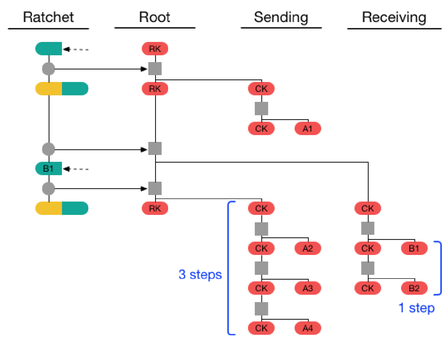

Figure 1. The Double Ratchet's two-stage process.

The Verification Challenge

The Double Ratchet's stateful, asynchronous nature makes security analysis **exceptionally difficult**. Simple testing cannot cover the enormous state space created by lost, duplicated, or reordered messages. A formal proof is needed to guarantee security against an active attacker who can manipulate the network. A single logic error (e.g. handling out-of-order messages) could lead to key reuse and a catastrophic failure of confidentiality.

From Types to Theorems (The F* Method)

As discussed in the "Project Everest" talk [3], formal methods treat software correctness as a mathematical proof. Instead of viewing a type as just a memory layout ('int is 4 bytes'), we view it as a logical proposition.

- **Dependent Types:** Types can depend on *values*, encoding invariants into the type system. [4] A function can have a type that requires, for example, a buffer that is *proven* to be 32 bytes long. This moves runtime error checks into compile-time guarantees, making entire classes of bugs impossible.
- **Types as Theorems:** The **Curry-Howard Correspondence** shows that a proposition is analogous to a type, and a proof of that proposition is a program of that type. A function that successfully type-checks against a complex dependent type is a *formal proof* that the code behaves as specified. This is the foundation of verified software like HACL* [5].

What Verification Looks Like in Practice (Examples)

To make these ideas concrete, consider how verification tools can prove critical properties at the code level.

1. Proving Memory Safety: A common bug is using a cryptographic key of the wrong size. With dependent types (as in F*), we can make this a compile-time error.

```
// F* type for a byte array PROVEN to have 32 bytes
type key_buffer = b:array u8{Array.length b = 32}

// This function CANNOT be called with a wrong-sized
// buffer. The compiler enforces this proof.
val aes256_encrypt : key:key_buffer -> ... -> ...
```

2. Preventing Timing Attacks: Comparing secrets naively can leak information through timing. Formal methods can prove that a function's execution time does not depend on secret data.

```
(* VULNERABLE: Leaks timing data *)
let isEqual (secret, input) =
  for i in 0..15 do
    if secret.[i] != input.[i] then
      return false (* Exits early! *)
  return true

(* SECURE: The type system can prove this function
   is "constant-time" with respect to the secret. *)
val isEqual_ct : secret:bytes -> input:bytes -> bool
```

Methodology: Verifying Signal

To prove Signal's security, cryptographers applied this rigorous approach using automated tools:

1. **Model the Protocol:** The protocol's roles, cryptographic functions, and message flows were translated into a precise mathematical model.
2. **Model the Attacker:** A powerful adversary (Dolev-Yao model) was defined with **complete control over the network** to intercept, modify, and inject messages.
3. **Define Security Properties:** Goals like "confidentiality" were translated into formal logic, e.g., *"an attacker cannot distinguish the real message plaintext from random data."*
4. **Automated Proof:** Tools like **ProVerif** [8] and **Tamarin** [9] symbolically explored every possible state, searching for any violation of the defined properties.
5. **DONE (yay)!**

Key Findings & Analysis

The seminal machine-checked analysis by Cohn-Gordon, et al. [10] used the Tamarin prover to produce a formal proof that the Signal protocol is sound.

- **Proven Security:** They proved that Signal achieves its advertised security goals, including strong authentication and confidentiality (IND-CCA), along with forward and post-compromise security, within their powerful attacker model [10].
- **Catching Subtle Flaws:** An earlier analysis of its predecessor, TextSecure, found a minor but significant "Unknown Key-Share" (UKS) attack [11]. In a UKS attack, an adversary could trick a user into thinking they were communicating with one person while their messages were being read and forwarded by the attacker. This flaw, missed by informal analysis, was fixed in Signal.

The Next Frontier: Post-Quantum Signal

The rise of quantum computers threatens to break the elliptic-curve cryptography (ECC) that underpins Signal's current security [12]. To counter this, Signal has developed and deployed **PQXDH**, a post-quantum key agreement protocol [13].

- **The Threat:** Shor's algorithm, which can be ran on a sufficiently powerful quantum computer, can efficiently break ECC and RSA [12].
- **The Solution:** PQXDH uses a quantum-resistant key encapsulation mechanism, **ML-KEM** [14]. It combines this with the existing X3DH in a **hybrid design** [15]. The protocol remains secure even if one of the underlying cryptographic primitives is broken — it is at least as secure as X3DH alone.

Future Work: The Triple Ratchet

Announced in October 2025, the **Triple Ratchet** introduces the **Sparse Post-Quantum Ratchet (SPQR)** to run alongside the classic Double Ratchet, providing ongoing, quantum-safe Forward Secrecy and Post-Compromise Security [16].

- **Hybrid Security:** Keys from both the classic (ECDH) ratchet and the new post-quantum (ML-KEM) ratchet are mixed together. An attacker must break *both* cryptographic systems [15]!
- **Efficient Design:** To handle the large key sizes of post-quantum algorithms, SPQR uses erasure codes to break keys into small chunks, making transmission resilient to network loss without consuming excessive bandwidth, especially over unreliable networks [17, 16].
- **Verified From The Start:** The new protocol was co-designed with formal verification using tools like ProVerif. The implementation is continuously verified in CI by translating the **Rust** code to **F*** using `hax` to prove correctness and safety properties on every change [18, 16].

Interactive Demo

Scan the QR code to explore an interactive visualization of the Double Ratchet and its formal verification on any device.

`https://doubleratchetsignal.netlify.app/`

(Full references are available on the web page)