Formal Verification in Practice

Lessons from Signal's PQC Migration So Far

Rolfe Schmidt
Signal Messenger
rolfe@signal.org
EU PQC Migration Conference 2025

Signal’s experience (so far) with formal verification

1. Intro
2. ProVerif: From one-time verification to a design tool
3. Hax: Verify code, not just the design
4. Jasmin: Verify assembly, not just the code
5. There is more
6. Wrap-up

Cryptographic code is notoriously difficult to get right.

The PQC Migration is going to force new implementations and deployments.

In many cases complexity will increase.

Fear of complexity.

Fear of replacing hardened code.

Fear of fixing things that “aren’t broken”.

So the PQC Migration is a daunting challenge?

Partly.

But it is also an opportunity.

Takeaway:

Today we have tools we didn't have before.

If we use them, we will emerge more secure and more agile.

Takeaway:

But also: we can continue making the tools better.

And more people will use them.

Let me justify this with some stories from Signal’s experience.

Chapter 1: Verifying Design with ProVerif

X3DH and PQXDH: The Problem

Flashback Spring 2023

Problem: An attacker that can compute curve25519 logarithms could compute X3DH session secrets and Double Ratchet updates, learning all session secrets.

Scope: All user messages and media were at risk to a HNDL attack.

But it wasn't 2013: NIST post-quantum standardization was pretty far along.

X3DH →PQXDH: Adding Post-Quantum Security

Classic X3DH

Alex **Blake**

$IK_A \leftrightarrow SPK_B$
$EK_A \leftrightarrow IK_B$
$EK_A \leftrightarrow SPK_B$
(* $EK_A \leftrightarrow OPK_B$ *)

SK = KDF(DH_1 || DH_2 || DH_3 || DH_4)

100s of bytes
Handshake overhead

X3DH →PQXDH: Adding Post-Quantum Security

Classic X3DH

Alex **Blake**

$IK_A \leftrightarrow SPK_B$
$EK_A \leftrightarrow IK_B$
$EK_A \leftrightarrow SPK_B$
(* $EK_A \leftrightarrow OPK_B$ *)

SK = KDF(DH_1 || DH_2 || DH_3 || DH_4)

100s of bytes
Handshake overhead

PQXDH

Alex **Blake**

$IK_A \leftrightarrow SPK_B$
$EK_A \leftrightarrow IK_B$
$EK_A \leftrightarrow SPK_B$
(* $EK_A \leftrightarrow OPK_B$ *)

KEM.Encaps($PQPK_B$)
→ (SS, CT)

SK = KDF(DH_1 || DH_2 || DH_3 || DH_4 || **SS**)

1000s of bytes
Handshake overhead

X3DH →PQXDH: Adding Post-Quantum Security

Classic X3DH

Alex **Blake**

$IK_A \leftrightarrow SPK_B$
$EK_A \leftrightarrow IK_B$
$EK_A \leftrightarrow SPK_B$
$(^* EK_A \leftrightarrow OPK_B\ ^*)$

$SK = KDF(DH_1 \,||\, DH_2 \,||\, DH_3 \,||\, DH_4)$

100s of bytes
Handshake overhead

PQXDH

Alex **Blake**

$IK_A \leftrightarrow SPK_B$
$EK_A \leftrightarrow IK_B$
$EK_A \leftrightarrow SPK_B$
$(^* EK_A \leftrightarrow OPK_B\ ^*)$

$KEM.Encaps(PQPK_B) \rightarrow (SS, CT)$

$SK = KDF(DH_1 \,||\, DH_2 \,||\, DH_3 \,||\, DH_4 \,||\, \mathbf{SS})$

1000s of bytes
Handshake overhead

Key Design Principles

- **Minimal change:** Add ONE post-quantum key/ciphertext
- **PQ HNDL protection:** A MAJOR gain
- **No loss of ECDH security:** Don't remove effective security!
- **Acceptable cost:** Storage costs are significant, but worth it.

Classical
Quantum vulnerable

HNDL-Hybrid
Partially quantum resistant

Simple, Right?

- In summer 2023 we circulated a draft specification.
- One group modeled it using **ProVerif** and found attacks.
  - The attacks did not work on our deployed implementation.
  - But the specification was incomplete and could be deployed insecurely.
- This led to a collaboration where, within a couple of weeks we had:
  - A corrected protocol specification.
  - Machine verified security proofs that
    - The new protocol has all of the security the old one had
    - The new protocol provides HNDL protection

Let’s see what this looks like.

A ProVerif Process

We can describe our protocol in a language that is (mostly) intuitive to experienced developers.

But: to use it at full power requires an understanding of its internals.

```
let Initiator(i:client, IKA_s:scalar) =
   …
   (* receive from the server the pre key bundles *)
   in(server, (SPKB_p:point,SPKB_sig:bitstring,PQPKB_p:kempub,PQPKB_sig:bitstring));
   (* Verify the signatures *)
   if verify(IKB_p,encodeEC(SPKB_p),SPKB_sig) then
   if verify(IKB_p,encodeKEM(PQPKB_p),PQPKB_sig) then
   (
    let (CT:bitstring,SS:bitstring) = pqkem_enc(PQPKB_p) in

    new EKA_s:scalar;

     let EKA_p = s2p(EKA_s) in
     let DH1 = dh(IKA_s,SPKB_p) in
     let DH2 = dh(EKA_s,IKB_p) in
     let DH3 = dh(EKA_s,SPKB_p) in
     let SK = kdf(concat5(DH1,DH2,DH3,DH4,SS)) in

     event InitDone(i,r,true,OPKB_p,SPKB_p,PQPKB_p,SK);

       let ad = concatIK(IKA_p,IKB_p) in
       new msg_nonce: bitstring;
       let msg = app_message(i,r,msg_nonce) in
       let enc_msg = aead_enc(SK,empty_nonce,msg,ad) in
       (* Send Message *)
       out(server, (IKA_p,EKA_p,CT,  OPKB_p, SPKB_p, PQPKB_p, enc_msg))
   )
 ).
```

Building a Confidentiality Query in ProVerif

We can build queries asking ProVerif to prove things about the protocol.

The attacker could have compromised Alex before time i:

```
event(RespondDone(b,a,spk,pqpk,sk))@i && attacker(sk) ⇒
    (event(CompromiseIK(a))@j1 && j1 < i)
```

Building a Confidentiality Query in ProVerif

We can build queries asking ProVerif to prove things about the protocol.

Or if Alex is not corrupted, DH could be broken before i

```
event(RespondDone(b,a,spk,pqpk,sk))@i && attacker(sk) ⇒
    (event(CompromiseIK(a))@j1 && j1 < i)
    || (event(BrokenDH())@j1 && j1 < i)
```

Building a Confidentiality Query in ProVerif

We can build queries asking ProVerif to prove things about the protocol.

Or if Alex is not corrupted and DH is not broken, Blakes SPK could be corrupted before i:

```
event(RespondDone(b,a,spk,pqpk,sk))@i && attacker(sk) ⇒
    (event(CompromiseIK(a))@j1 && j1 < i)
    || (event(BrokenDH())@j1 && j1 < i)
    || (event(CompromiseSPK(b, spk)@j1 && j1 < i)
```

Sometimes ProVerif proves security.

Sometimes it finds an attack

And when it does it gives you a step-by-step trace showing you how to perform the attack.

1. Using the function info_x25519_sha512_kyber1024 the attacker may obtain info_x25519_sha512_kyber1024.
attacker(info_x25519_sha512_kyber1024).

2. Using the function zeroes_sha512 the attacker may obtain zeroes_sha512.
attacker(zeroes_sha512).

…

19. By 18, the attacker may know
(SMUL(IK_s_1,G),SMUL(EKA_s_1,G),penc(SMUL(SPKB_s_3,G),ss_1),SMUL(y,G),SMUL(SPKB_s_2,G),SMUL(SPKB_s_3,G),aead_enc(hkdf(concat(ff_x25519,concat5(SMUL(IK_s_1,SMUL(SPKB_s_2,G)),SMUL(IK_s_2,SMUL(EKA_s_1,G)),SMUL(EKA_s_1,SMUL(SPKB_s_2,G)),SMUL(EKA_s_1,SMUL(y,G)),ss_1)),zeroes_sha512,info_x25519_sha512_kyber1024),empty_nonce,app_message(a,b,msg_nonce_1),concatIK(SMUL(IK_s_1,G),SMUL(IK_s_2,G)))).
Using the function 3-proj-7-tuple the attacker may obtain penc(SMUL(SPKB_s_3,G),ss_1).
attacker(penc(SMUL(SPKB_s_3,G),ss_1)).

20. By 19, the attacker may know penc(SMUL(SPKB_s_3,G),ss_1).
Using the function weakECasKEM the attacker may obtain ss_1.
attacker(ss_1).

…

Details!

Protocol description: https://signal.org/docs/specifications/pqxdh/

[BJKS24]:
https://www.usenix.org/conference/usenixsecurity24/presentation/bhargavan

ProVerif at Signal Today

Now we use ProVerif as part of our design process

- For our new Sparse Post-Quantum Ratchet
  - We iterated through at least a dozen designs
  - and used ProVerif to determine fine-grained security
  - that helped us analyze and compare candidates
  - and prove that our final protocol met its goals.
- Future Handshake Protocols
  - There is more beyond PQXDH.
  - We use ProVerif to evaluate improvements and prove they meet new goals.
- We’re exploring its use more broadly to model new features.

So your design is good. What about the code?

Verification needs to become part of engineering.

Chapter 2: Verifying Implementations with Hax

High Assurance Code

Hax (https://hax.cryspen.com/) is a tool for high assurance translations of a large subset of Rust into formal languages such as F*, Lean, EasyCrypt, Coq, and ProVerif.

Verifying our Sparse Post Quantum Ratchet

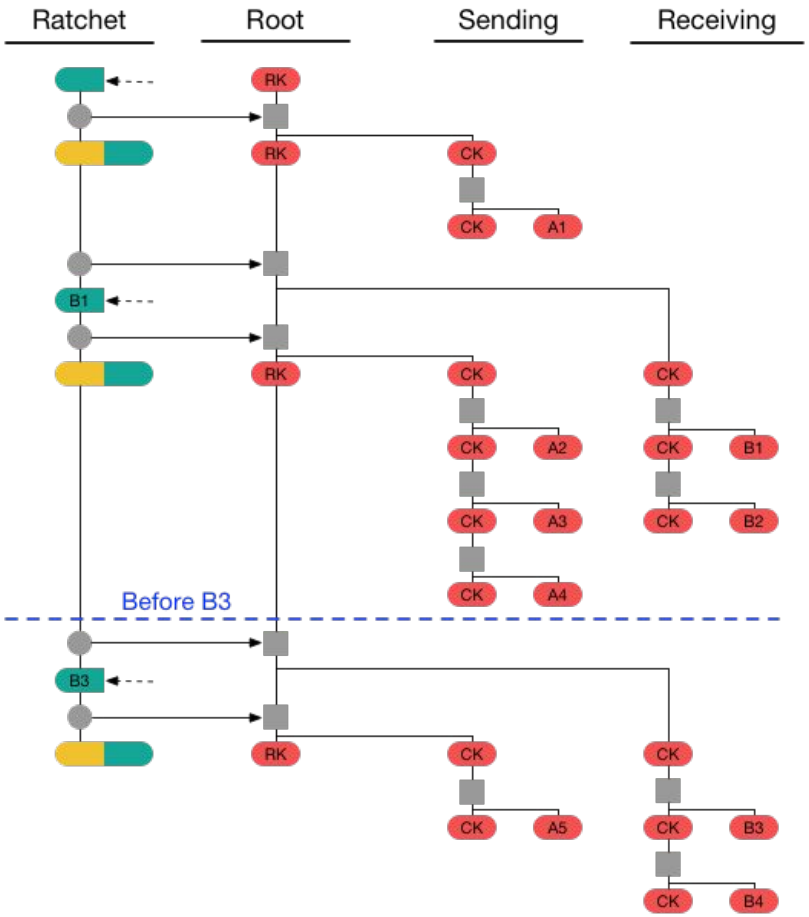
In September 2025 Signal announced deployment of the Sparse Post-Quantum Ratchet, a major step in the Post-Quantum migration of the Signal Protocol.

The Signal Protocol is the core of security throughout our application and there is no room for error.

We have to move fast, but we can’t break things.

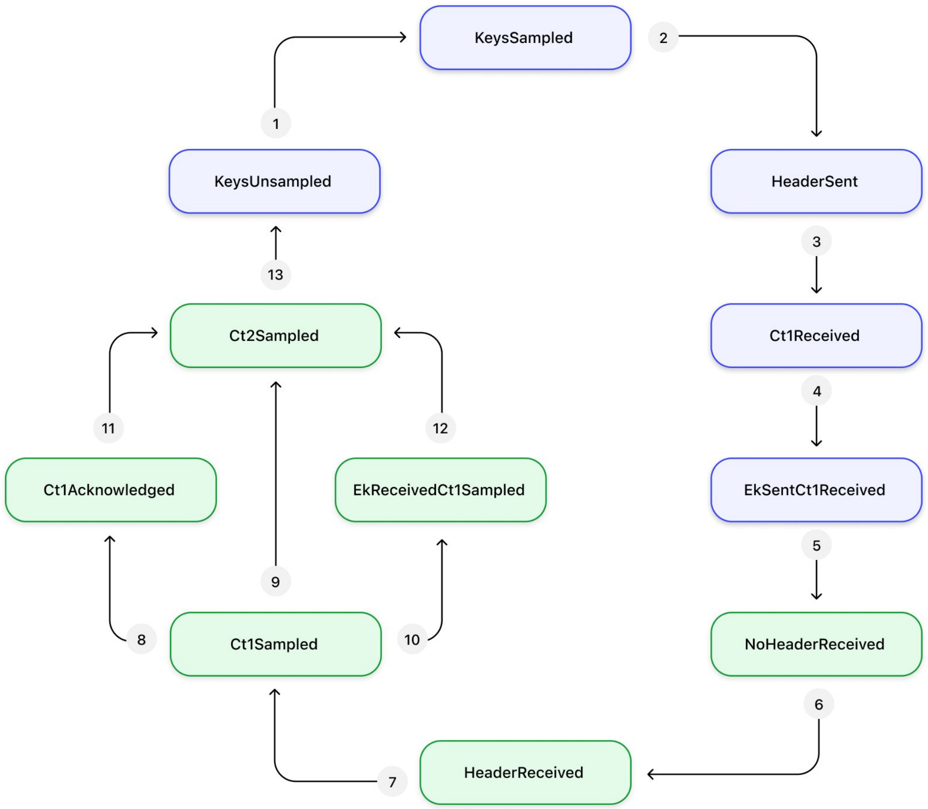
The Double Ratchet was Complex

- **Forward Secrecy:** It updates keys every time a message is sent so if a device is compromised it leaks nothing about old messages.
- **Post-Compromise Security:** It performs new key exchanges on every round trip to “heal”
- **Immediate Decryption:** It still works if messages are dropped or out of order

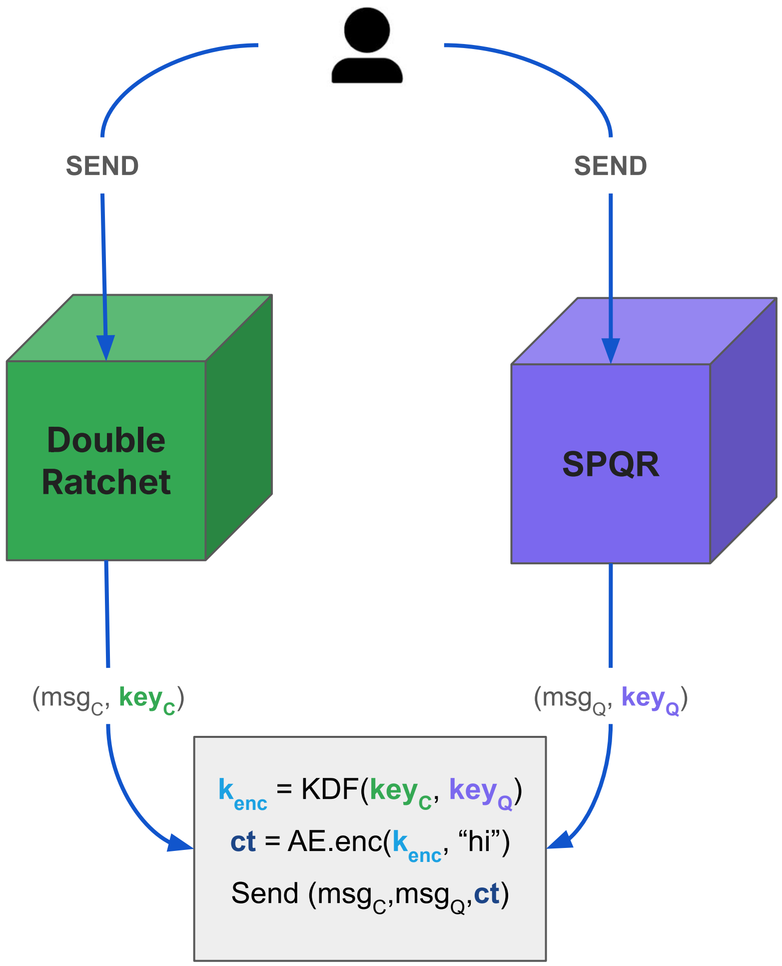


SPQR is All That and More

- PQ Keys and Messages are large, but we limit per-message overhead to 40 bytes.
- We use **erasure codes** and **internal KEM structure** to create an optimized, robust continuous key agreement.
- Double Ratchet had a 2-state machine, SPQR has an 11-state machine.
- Double Ratchet had 1 symmetric ratchet per epoch, SPQR has 2.



And we run it in parallel with the Double Ratchet



ProVerif was a key tool in the design process.

But how can we be sure we got the implementation right?

Hax in SPQR

We implemented SPQR in Rust (https://github.com/signalapp/SparsePostQuantumRatchet).

Every time an engineer pushes a change to version control:

- Hax translates the Rust code to F*
- With F* we prove that the implementation is panic free and correct[1].
- We fill the code with security-relevant assertions so this proves our assumptions are true.
- **If the proofs fail, the build fails**
    - If the build fails, the change cannot be merged.

Hax in SPQR

It’s not just our code, it’s our dependencies too:

- We use libcrux for our cryptographic primitives.
- Libcrux has been formally verified with the hax toolchain

Looking forward

- We can translate the code into Lean or ProVerif
- There we can prove **security** in addition to **runtime safety** and **correctness**.

This is the future we need to build.

So your code is good, but what about the binary?

Machines don’t execute C or Rust.

They execute machine instructions.

We need assurance they are doing what we need them to do.

Chapter 3: (Fast) Verified Assembly with Jasmin and EasyCrypt

Problem: Code Running in the Cloud

Whenever possible, Signal designs services with cryptographic guarantees that the servers do not need to be trusted.

For two services - Contact Discovery and Secure Value Recovery - we need to handle sensitive data on cloud servers.

To do this securely, we run these services inside Intel SGX TEEs.

Problem: Code Running in the Cloud

TEEs offer strong guarantees against *software* attacks (acknowledging recent developments).

But they explicitly do not protect against **side channel attacks** where timing, memory access, or architectural optimizations can leak information about critical secrets.

So we write **constant time**, **data oblivious** code for these services.

No branch or memory access depends on secret data - or on data derived from secret data.

Well the code is constant time and data oblivious.

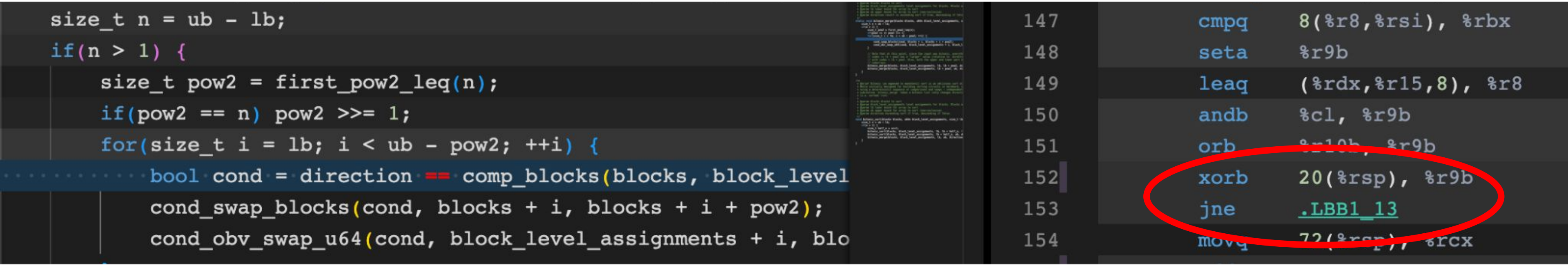
What about the binary?

Compilers are too good.

For all of our efforts to write careful constant time code…

…optimizing compilers can see through it and help us speed things up…

…introducing leaks that we can only find by analyzing the assembly code.



Jasmin and EasyCrypt

But again, there are tools we can use.

- Libjade is a library of cryptographic primitives
  - implemented in Jasmin
  - Verified with EasyCrypt
- We are integrating it into our services to have high assurance that app-to-enclave secure channels can leak nothing to our hosts.
- And it is **fast**.

We are thinking much bigger than that. Stay tuned.

There is much more.

We use **TLA+** to analyze concurrent systems.

Tamarin has been effective at analyzing Signal.

We’re looking to use it more.

Lean, Verus, OWL,
Aeneas, …

Conclusions

We have powerful tools that are valuable in real deployments **today**.

If we make them part of the PQC migration process we’ll have done much more than get quantum safety.

But the edges are rough.

Learning curves are steep.

Things change fast.

To realize this potential, we need to keep improving the tools and encourage their use.

Thank you!

rolfe.1729

Scan this QR code with your phone to chat with me on Signal.

rolfe@signal.org