# High Assurance Post Quantum Cryptography

Karthikeyan Bhargavan

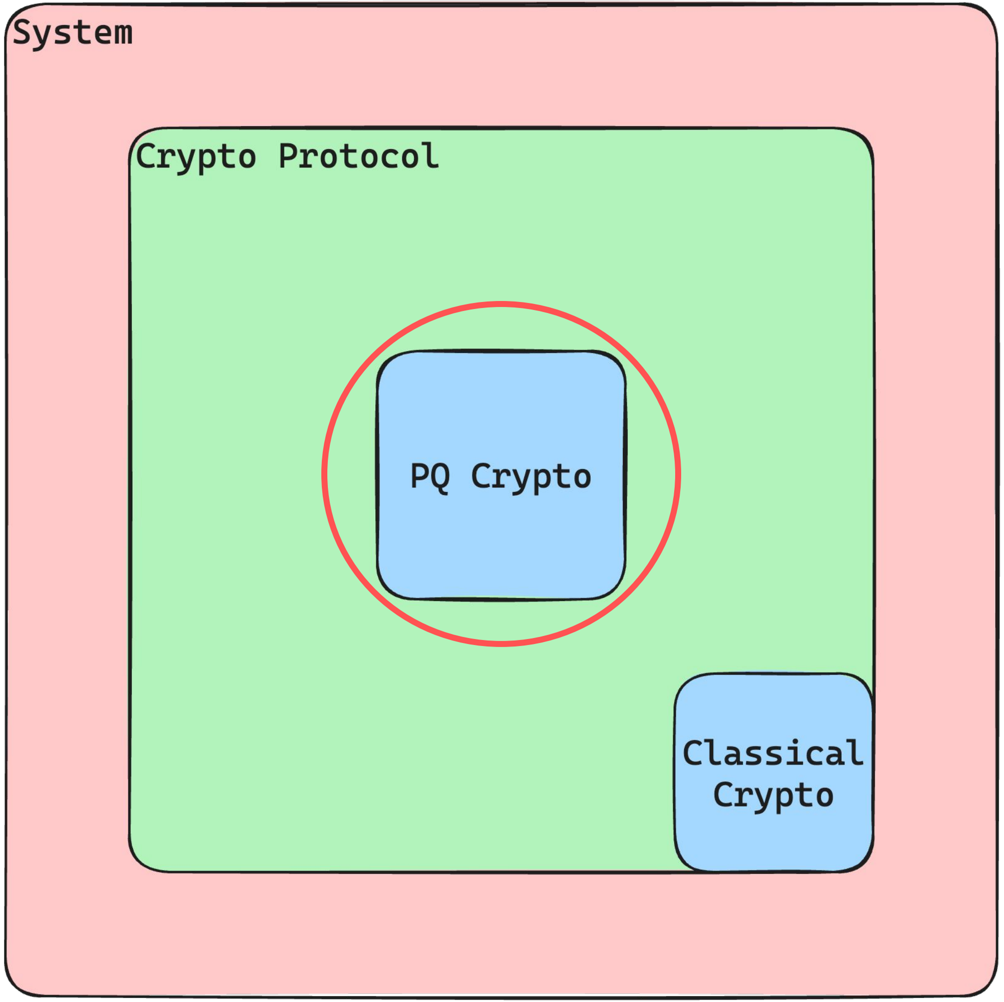
Joint work with Rolfe Schmidt (Signal), Charlie Jacomme (Inria), Franziskus Kiefer (Cryspen), Goutam Tamvada (Cryspen), Lucas Franceschino (Cryspen), Jonathan Protzenko (MSR), …

MatchPoints 2024, Aarhus

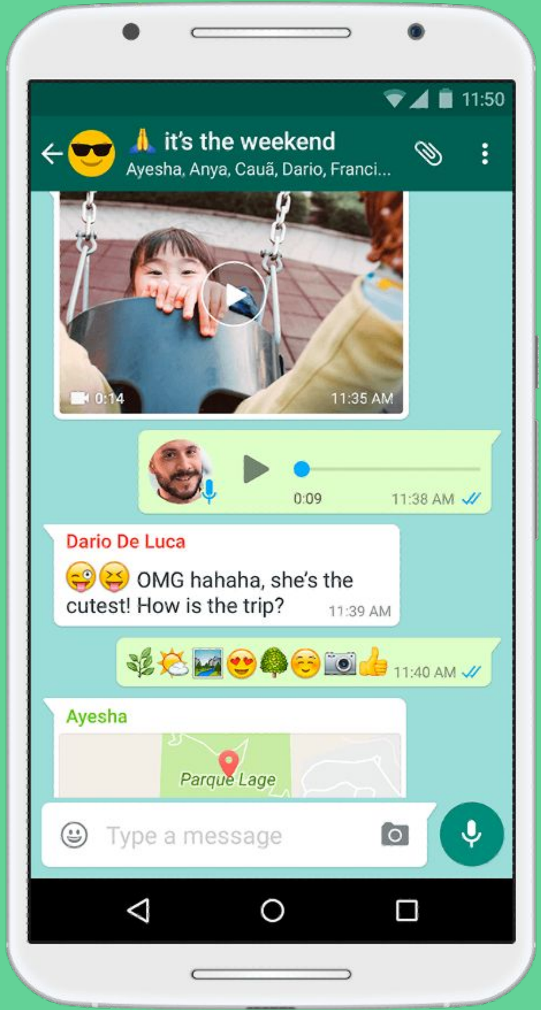
CRYSPEN

Formal verification can **speed development** and **clarify security** of real world systems.

This is important as many applications are being updated to provide **Post-Quantum** security.


System
Crypto Protocol
PQ Crypto
Classical
Crypto

Let’s see how this process worked with the **PQ transition** of **Signal Messenger**



# The Signal Messaging Protocol

# The Signal Protocol

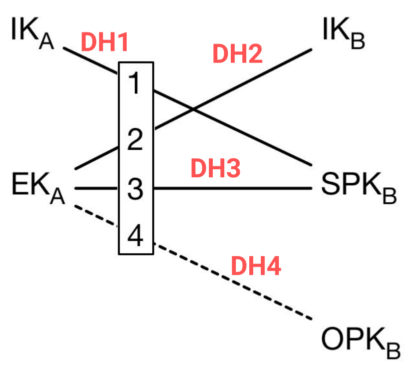
Two parts:

- X3DH handshake
- Double Ratchet for continuous key agreement

Important security guarantees:

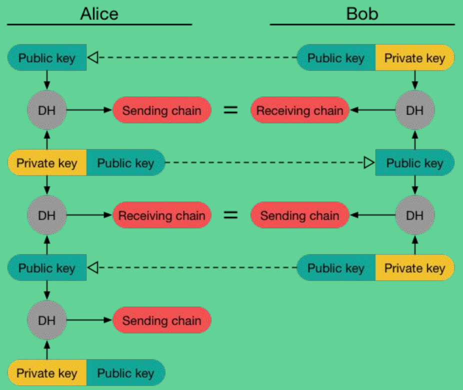
- Confidentiality
- Mutual authentication
- Post-compromise security
- Forward secrecy
- Deniability

## X3DH




**SK** = KDF(**DH1** || **DH2** || **DH3** || **DH4**)

## Double Ratchet



# The Signal Protocol

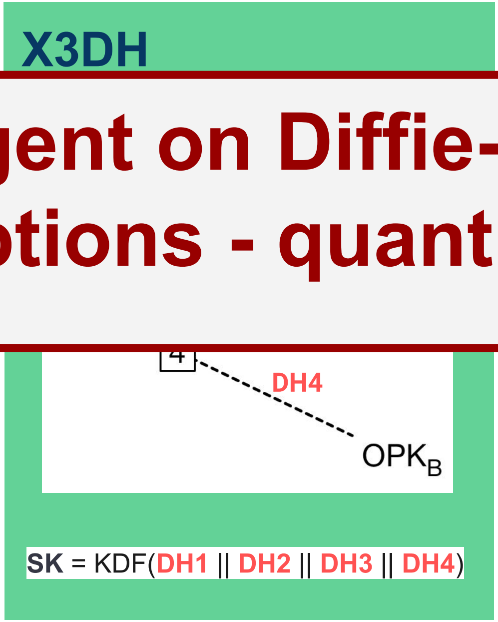
Two parts:

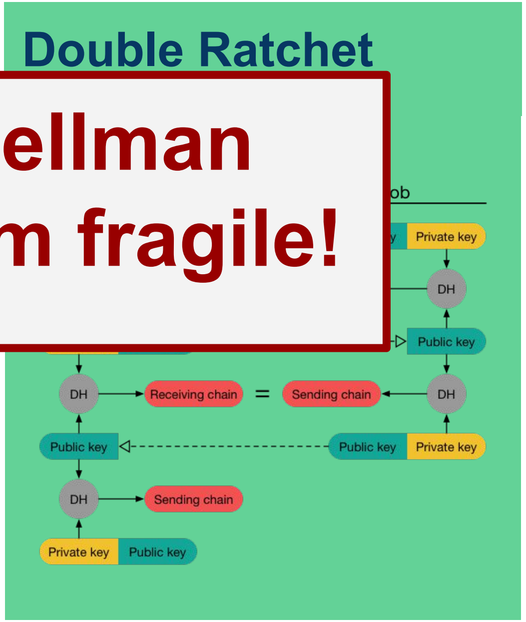
- X3DH handsl
- Double Ratch
  continuous k

Important security

- Confidentiality
- Mutual authentication
- Post-compromise security
- Forward secrecy
- Deniability

**Contingent on Diffie-Hellman assumptions - quantum fragile!**



Signal is vulnerable to any future discrete logarithm solver - quantum *or* classical.

**Harvest Now, Decrypt Later** (HNDL) attacks:

Messages sent today are vulnerable to quantum attackers tomorrow

# The PQXDH Key Agreement Protocol

# PQXDH Protocol Requirements

- Provide HNDL protection against future DL solvers
- No loss of current DH-based security guarantees

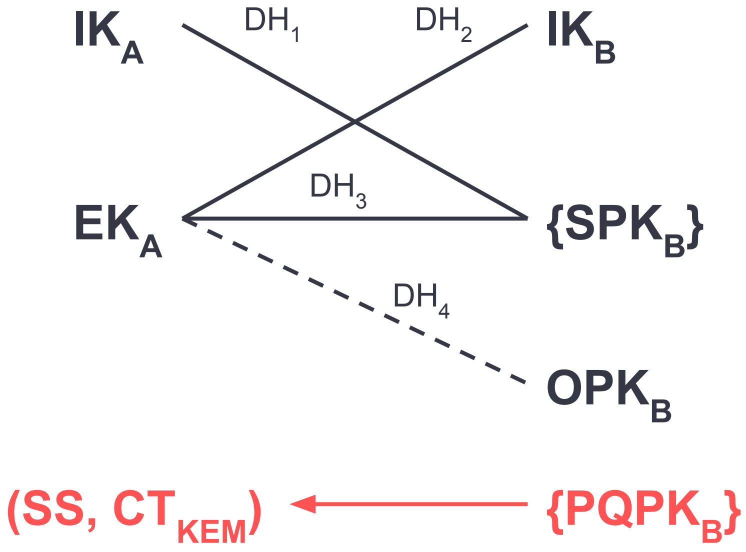
Non-goal: Protect against active quantum attackers

To achieve this we need to add PQ crypto to the X3DH handshake.

**A simple idea:**

Take X3DH and
add in a PQ-KEM
encapsulated shared secret.

# PQXDH Design




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

After computing **SK**, Alice sends to Bob:

- **(C, CT$_{KEM}$, EK$_A^{PK}$)** where
- **C** = AEAD.Enc(**SK**, *msg*, AD = **IK$_A^{PK}$** || **IK$_B^{PK}$**)

Bob processes the message by:

- Using their EC keys to compute the **DH**'s
- Using their KEM key to decapsulate **SS**
- Computing **SK**
- Computing AEAD.Dec(**SK**, **C**, AD)

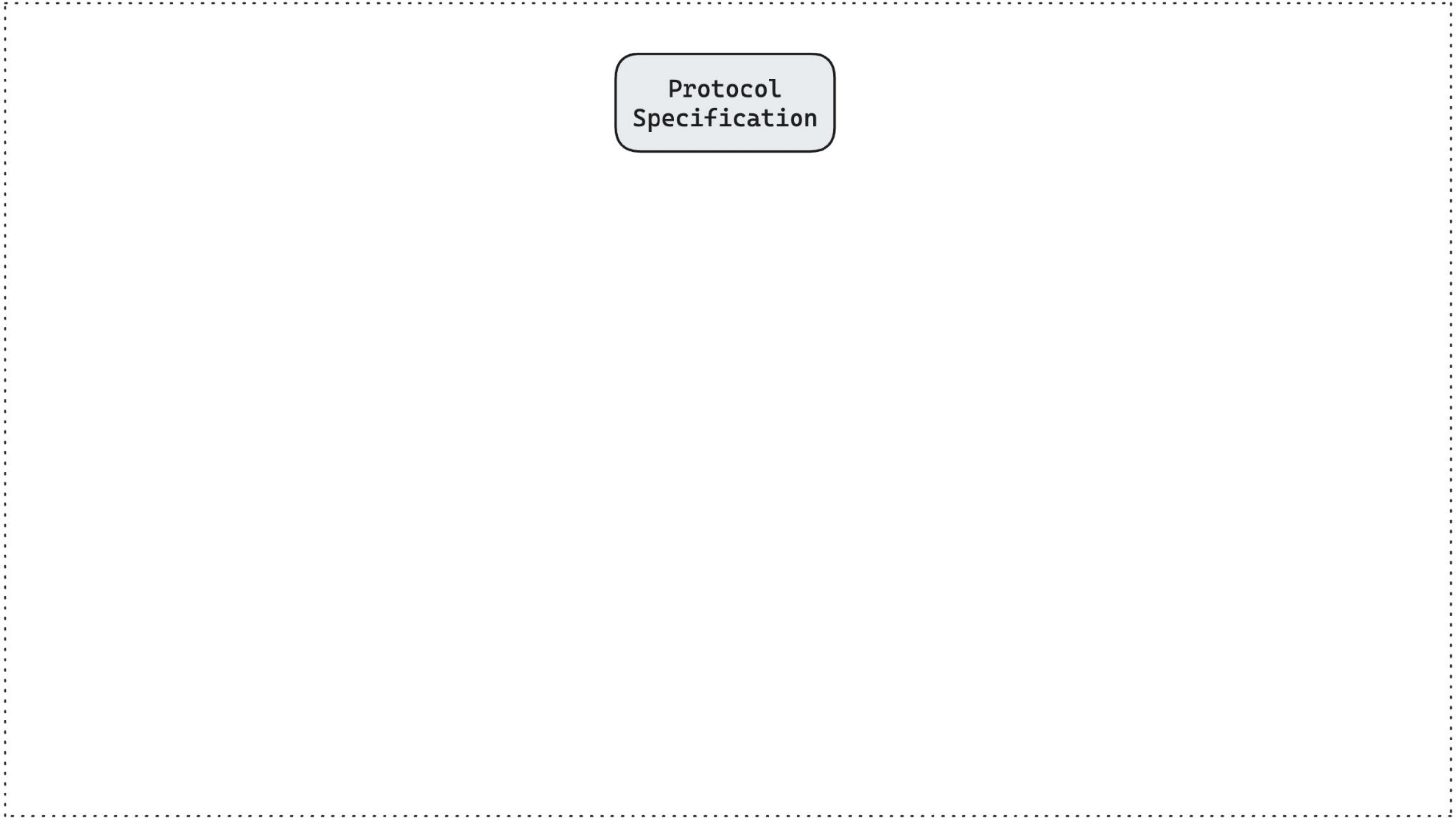
If the decryption succeeds, we have key agreement.

Does PQXDH achieve its goals?

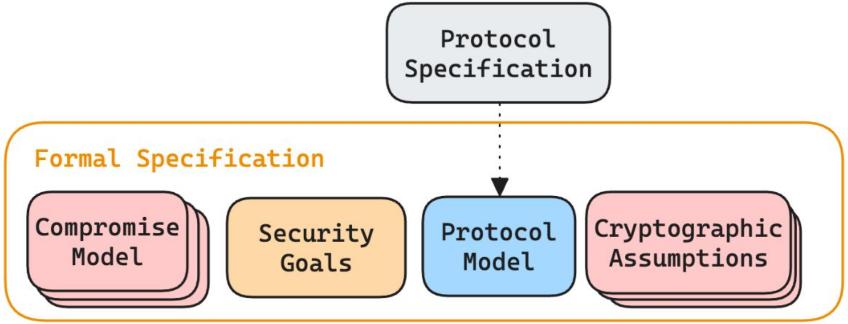
We need to **formally verify** it.

# Formally Modelling PQXDH

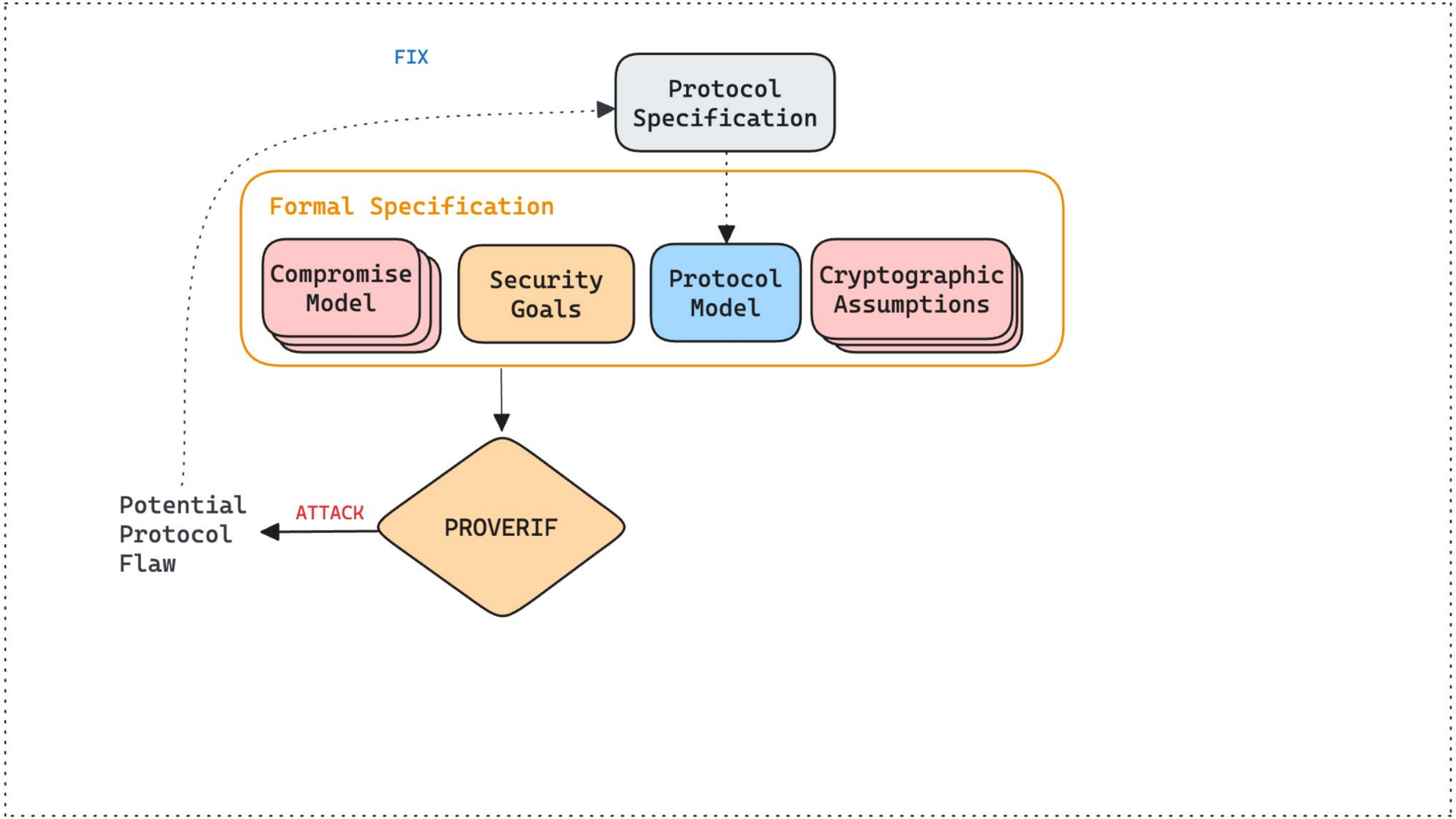
# Our Formal Verification Methodology



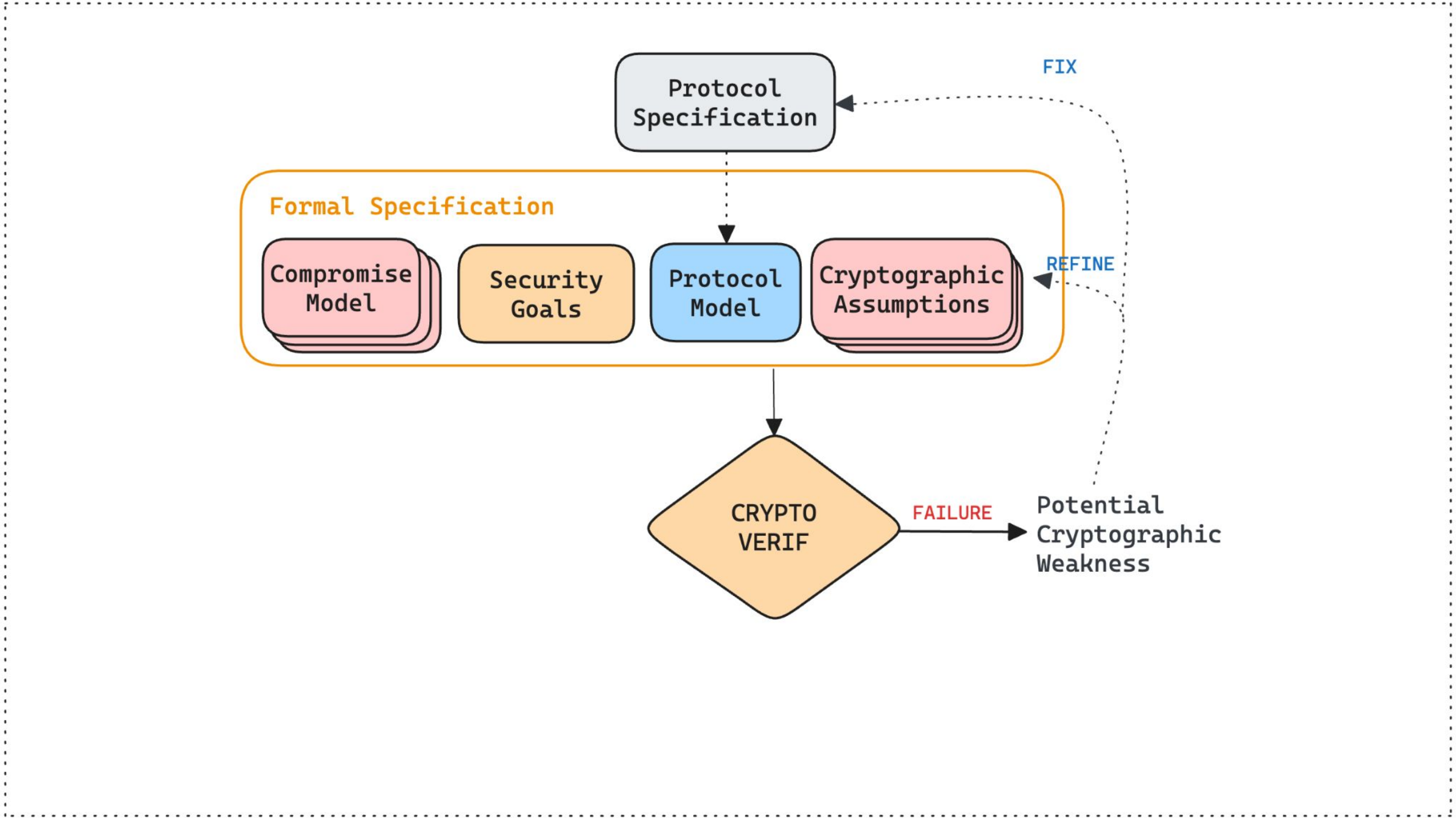
# Our Formal Verification Methodology



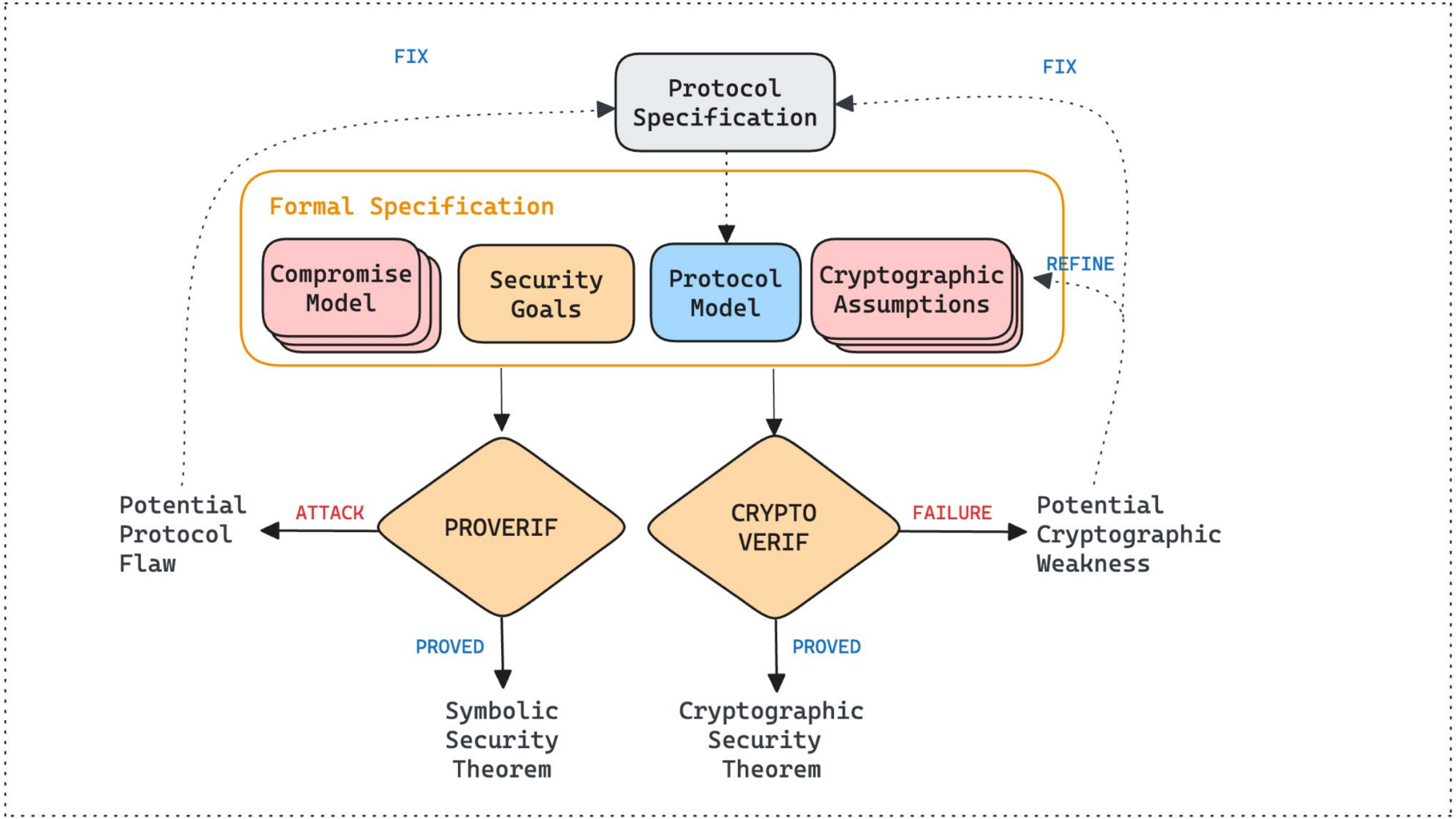
# Our Formal Verification Methodology



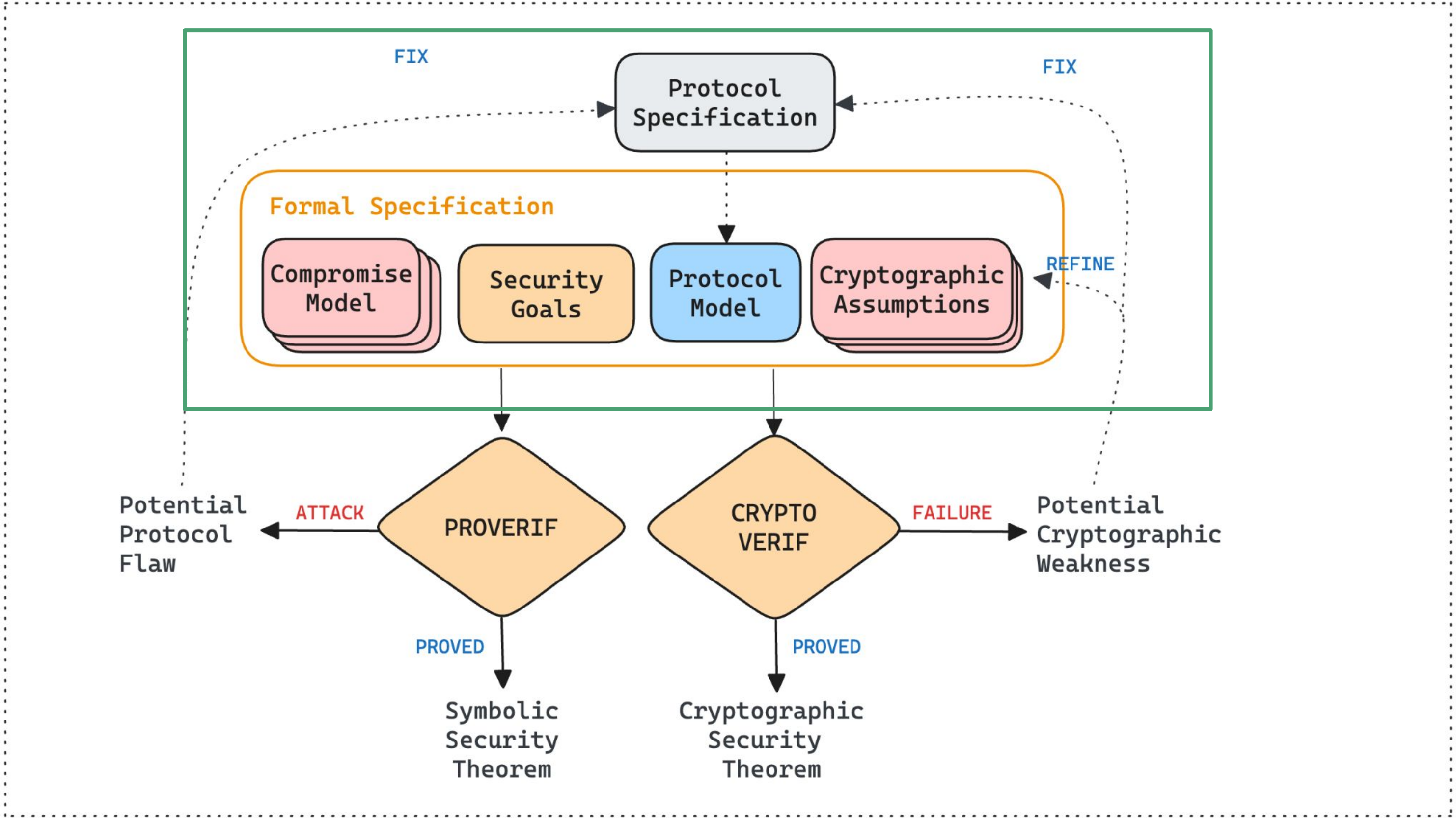
# Our Formal Verification Methodology



# Our Formal Verification Methodology



# Our Formal Verification Methodology



# What We Model

## Single Message PQXDH Protocol

- Arbitrary number of PQXDH endpoints
- Any endpoint can play any role
- (Out-of-Band) Identity Key Verification
- Untrusted Key Distribution Server

## Compromise Scenarios

- Identity keys can be leaked at any time
- OPK, EK, and PQPK can be leaked for certain security goals
- Quantum adversary has explicit power to break all DH primitives

```
let Initiator(i:client, IKA_s:scalar) =
    (* Download Responder Keys *)
    ...

    (* Verify the signatures *)
    if verify(IKB_p,encodeEC(SPKB_p),SPKB_sig) then
    if verify(IKB_p,encodeKEM(PQPKB_p),PQPKB_sig) then

    (* PQXDH Key Derivation*)
    let IKA_p = s2p(IKA_s) in
    let (CT:bitstring,SS:bitstring) =
        pqkem_enc(PQPKB_p) in (* PQ-KEM Encap *)
    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 DH4 = dh(EKA_s,OPKB_p) in
    let SK = kdf(concat5(DH1,DH2,DH3,DH4,SS)) in

    (* Send Message *)
    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

    out(server, (IKA_p,EKA_p,CT,OPKB_p,
                 SPKB_p,PQPKB_p,enc_msg))
```

# Symbolic Analysis with ProVerif

### Symbolic (Dolev-Yao) Crypto Model

- “Perfect” crypto primitives
- Unbounded number of sessions
- Previously used for Signal, TLS 1.3, ...

### Quantum Adversary Model

- Adversary can invert DH

### Security Analysis

- Queries for authentication and secrecy
- Fully automated analysis
- Finds attacks or establishes a theorem
- Easy to quickly test fixes

```
(* Post-Quantum Forward Secrecy Query *)
query A, B, spk, pqpk, sk, i, j;
      event(BlakeDone(A,B,spk,pqpk,sk))@i
            ⇒ not(attacker(sk))
              | (event(LongTermComp(A))@j & j < i)
              | (event(QuantumComp)@j & j < i)
```

**Attack Trace:**

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).

3. We assume as hypothesis that
attacker(a).

4. We assume as hypothesis that
attacker(b).

5. The message b that the attacker may have by 4 may be received at input {2}.
So the entry identity_pubkeys(b,SMUL(IK_s_2,G)) may be inserted in a table at i
table(identity_pubkeys(b,SMUL(IK_s_2,G))).

# Computational Proofs with CryptoVerif

**Computational Crypto Model**

- Precise Cryptographic Assumptions
- Probabilistic Polynomial-Time Adversary

**Quantum Adversary Model**

- Adversary can (passively) break DH
- Uses new Post-Quantum Soundness results for CryptoVerif proofs

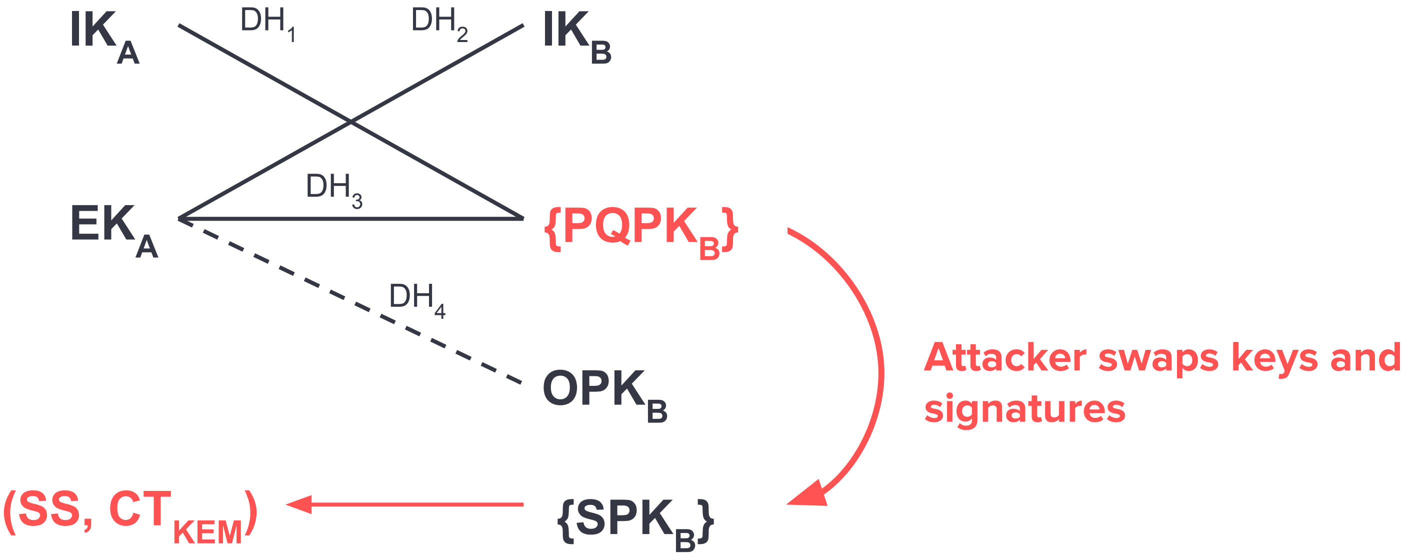
**Security Analysis**

- Queries for authentication and secrecy
- Game-based machine-checked proofs
- Similar guarantees to pen-and-paper proofs
- Requires manual guidance

```
proof {
crypto uf_cma_corrupt(sign) signAseed;
out_game "g1.cv" occ;

insert before "EKSecA1 <-R Z" ...
insert after "RecvOPK(" ...
out_game "g11.cv" occ;

insert after "OH_1(" ...
crypto rom(H2);
out_game "g2.cv" occ;

insert before "EKSecA1p <-R Z" ...
insert after "RecvNoOPK(" ...
out_game "g12.cv"occ;

insert after "OH(" ...
crypto rom(H1);
out_game "g3.cv";

crypto gdh(gexp_div_8) ...
crypto int_ctxt(enc) *;
crypto ind_cpa(enc) **;
out_game "g4.cv";

crypto int_ctxt_corrupt(enc) r_23;
crypto int_ctxt_corrupt(enc) r_50;
success
}
```

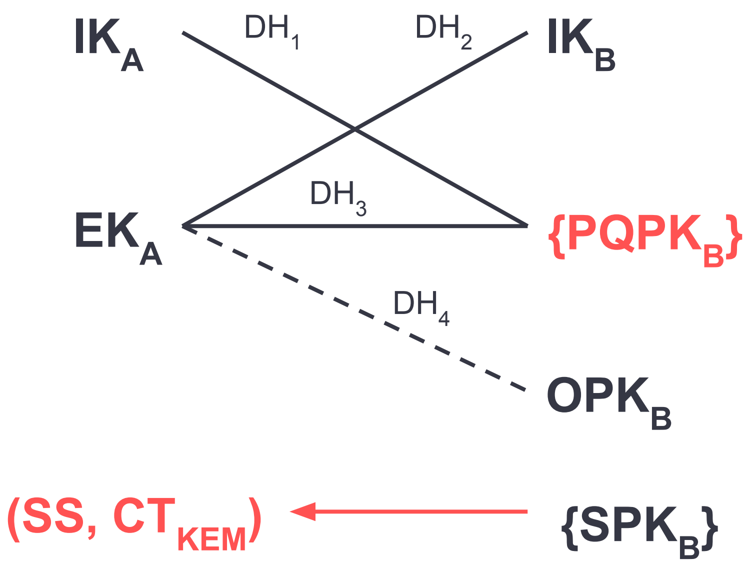
# Finding and Confirming Weaknesses

# Key Confusion Attack




$$SK = KDF(DH_1 \parallel DH_2 \parallel DH_3 \parallel DH_4 \parallel SS)$$

# Key Confusion Attack




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

Now Alex computes :
$(SS, CT) = KEM.Encaps(SPK_B^{PK})$

Without further assumptions about KEM **this is an insecure computation.**

Given **CT** the attacker can now compute **SS**.

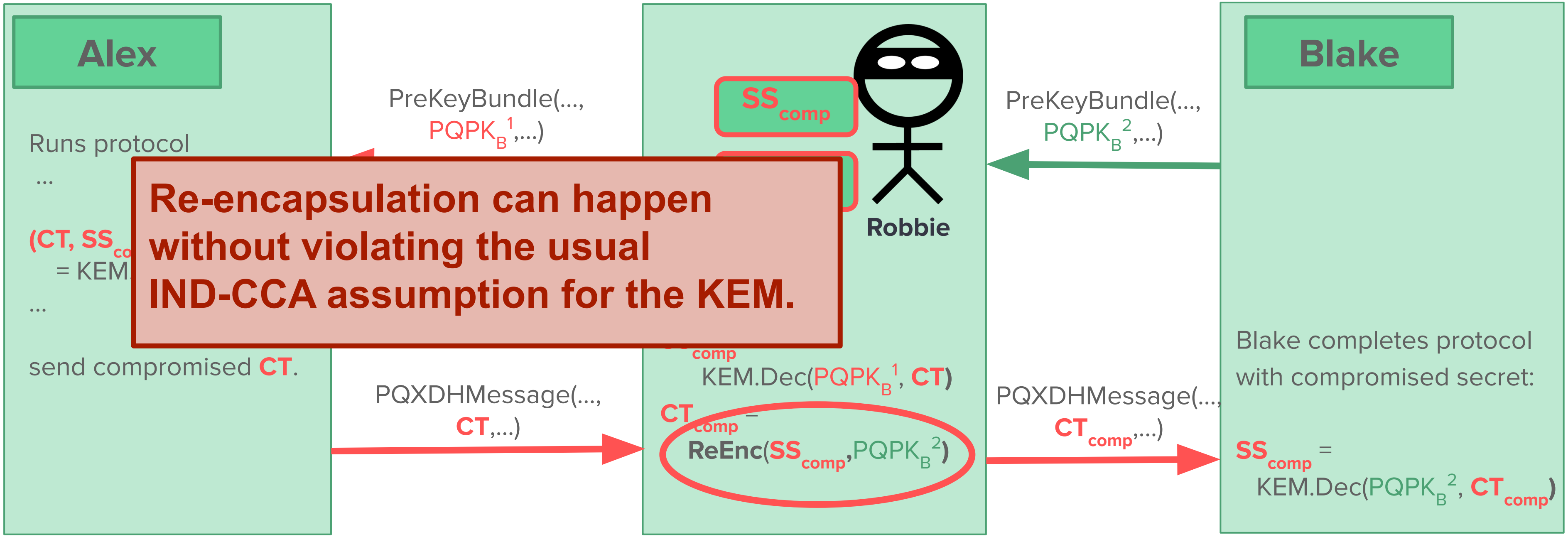
**We lose PQ security.**

This is representative of a general class of cross-protocol attacks between classical and PQ crypto.

**Fix:** Ensure all key encodings have disjoint co-domains.

# KEM Re-encapsulation Vulnerability

Attacker re-encrypts a PQ-KEM ciphersuite for another key to confuse the recipient and break session independence



# A New Revision of PQXDH

# The Deployed Signal Protocol was Secure

**The open-source messenger app was never vulnerable:**

- No Key Confusion:
  Signal's key encodings have disjoint co-domains
- No KEM Re-Encapsulation:
  Kyber public keys are hashed into the KEM shared secret

**But we still want to strengthen the protocol specification.**

# PQXDH Version 2 (one month later)

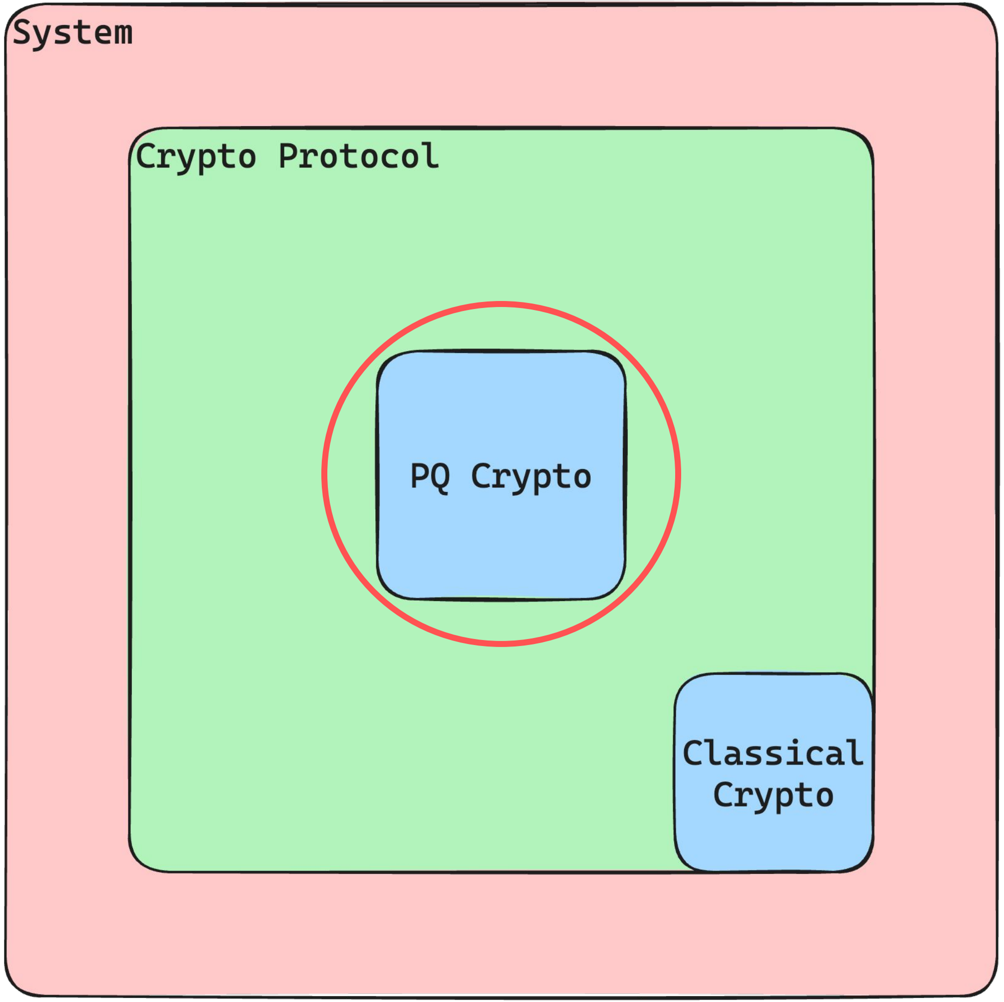
The findings led to a new revision of the protocol:

- We added **AEAD** as a parameter and required it to be post-quantum **IND-CPA** and **INT-CTXT**
- Added description of key identifier use — **Not security relevant**
- Restricted the ranges of encodings to be disjoint — **Prevent Key Confusion Attack**
- Added **$PQPK_B^{PK}$** to AD when it isn't contributory to the KEM

**Prevent KEM Re-encapsulation Attack**

With these changes **we proved security theorems** that PQXDH meets its security requirements in the **symbolic, computational, and PQ HNDL models.**

# But is the Signal *Implementation* Secure?


System
Crypto Protocol
PQ Crypto
Classical
Crypto

# FIPS 203 (Draft)

## Federal Information Processing Standards Publication

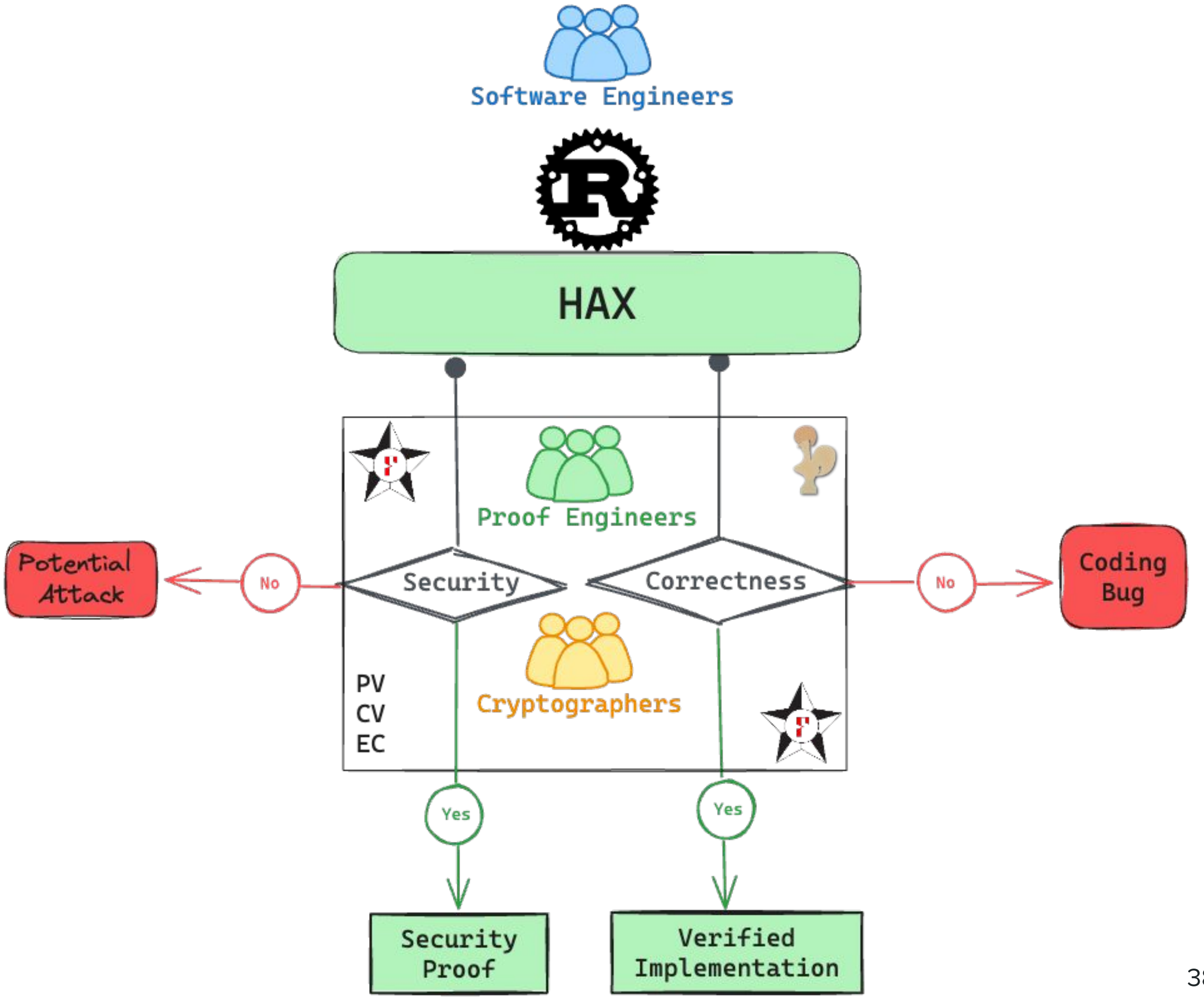
# Module-Lattice-based Key-Encapsulation Mechanism Standard

**Category: Computer Security** **Subcategory: Cryptography**

# Formally Verifying the new ML-KEM cryptographic implementation

Using the hax toolchain

# hax verification toolchain

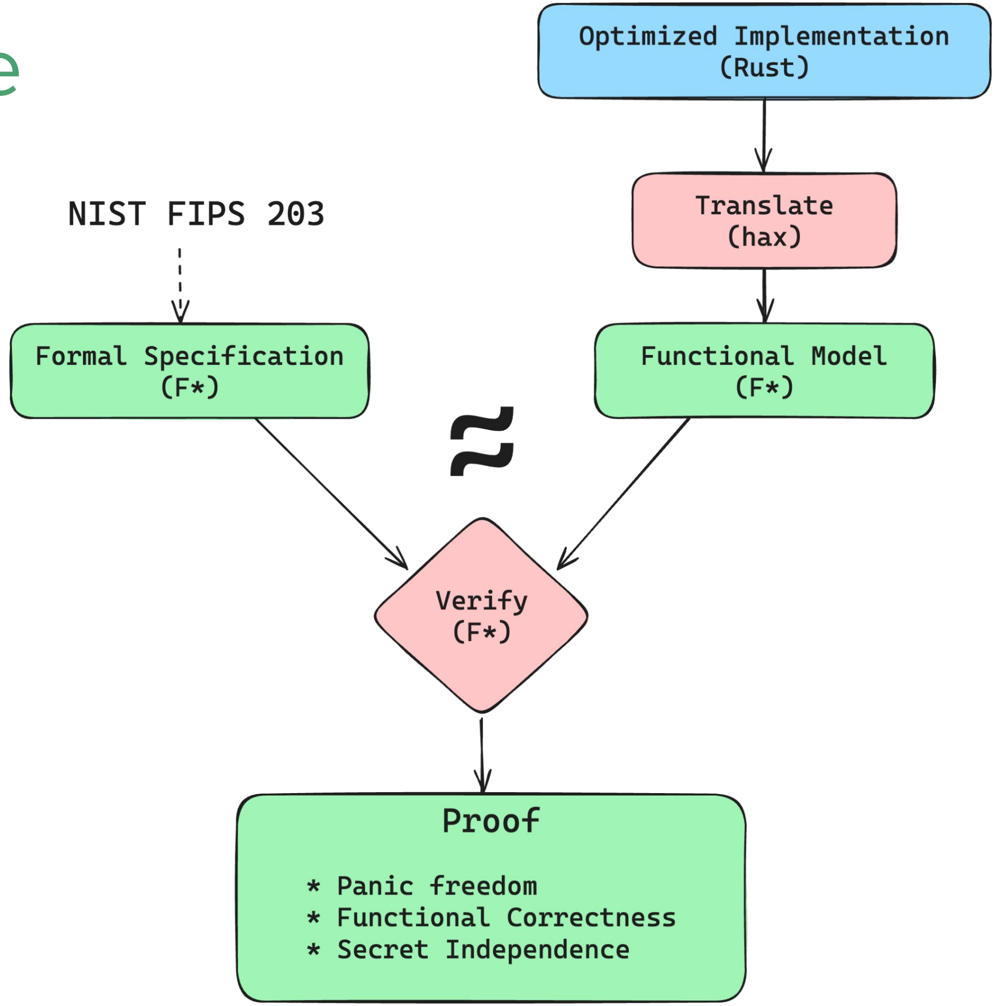



CRYSPEN

AARHUS UNIVERSITY

Inria

# Verifying Rust Code with hax and F*



# Writing Crypto Code in Rust

```
pub(crate) fn barrett_reduce(input: i32) -> i32 {
    let t = (i64::from(input) * 20159) + (0x4_000_000 >> 1);
    let quotient = (t >> 26) as i32;
    let remainder = input - (quotient * 3329);
    remainder
}
```

**Barrett Reduction:** computes **input % 3329**

(in constant time)

# Potential Panics in Rust Code

```
pub(crate) fn barrett_reduce(input: i32) -> i32 {
    let t = (i64::from(input) * 20159) + (0x4_000_000 >> 1);
    let quotient = (t >> 26) as i32;
    let remainder = input - (quotient * 3329);
    remainder
}
```

These arithmetic operations may overflow or underflow causing the code to panic at run-time

# Proving Panic Freedom and Correctness in F*

```
val barrett_reduce (input: i32_b (v v_BARRETT_R))
    : Pure (i32_b 3328)
    (requires True)
    (ensures fun result ->
        v result % v Libcrux.Kem.Kyber.Constants.v_FIELD_MODULUS
      = v  input %v Libcrux.Kem.Kyber.Constants.v_FIELD_MODULUS)
```

**Expected behaviour:** `result ≈ input % 3329`

# Enforcing Secret Independence

**Static analysis of forbidden operations**

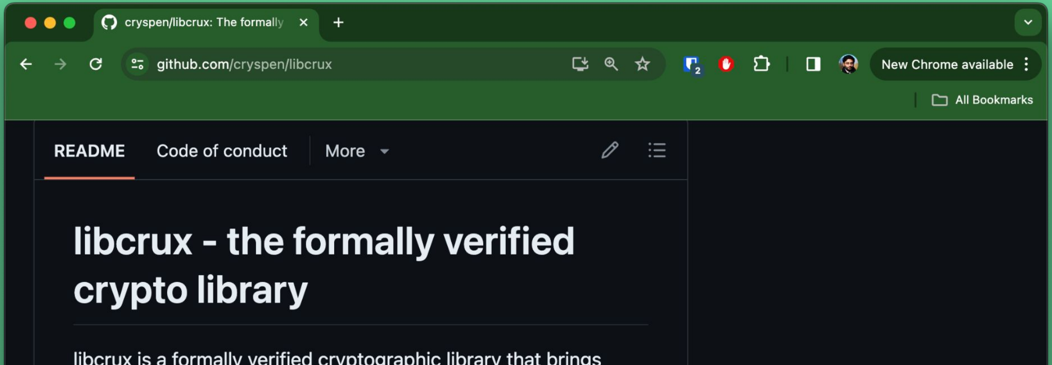
- **arithmetic operations** with input-dependent timing (e.g. division) over secret integers
- **comparison** over secret values
- **branching** over secret values
- **array** or vector **accesses** at secret indices

# A New Timing Vulnerability in ML-KEM libraries

```
void poly_tomsg(uint8_t msg[KYBER_INDCPA_MSGBYTES], const poly *a)
{
  unsigned int i,j;
  uint16_t t;

  for(i=0;i<KYBER_N/8;i++) {
    msg[i] = 0;
    for(j=0;j<8;j++) {
      t  = a->coeffs[8*i+j];
      t += ((int16_t)t >> 15) & KYBER_Q;
      t  = (((t << 1) + KYBER_Q/2)/KYBER_Q) & 1;
      msg[i] |= t << j;
    }
  }
}
```

Bug in PQ-Crystals, PQ-Clean, ...
(also used in Signal)

We built an **optimized**, **portable**, formally **verified** implementation of ML-KEM in Rust and C



# Conclusion

- The PQ transition is about more than just swapping in PQ crypto.
- There are many potential pitfalls, as we found in PQXDH and ML-KEM

- Protocol verification can help find and prevent attacks in PQ protocols.
- Software verification can help find and prevent implementation bugs
- Verification can also justify new optimizations to improve performance

- Close collaboration between protocol designers, developers, and proof engineers can provide quick turnaround and help guide the transition