High Assurance Post Quantum Cryptography

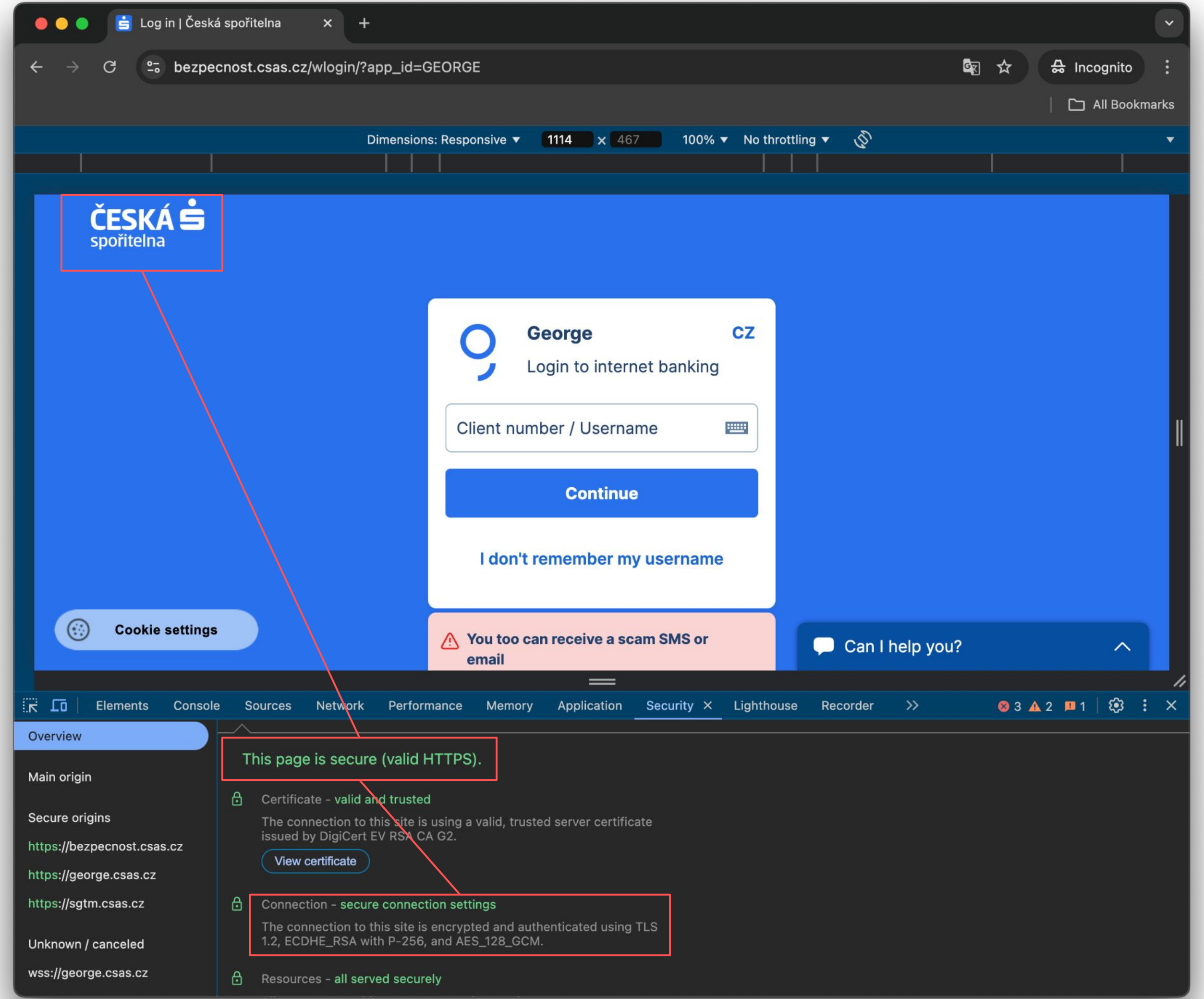
Karthikeyan Bhargavan

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

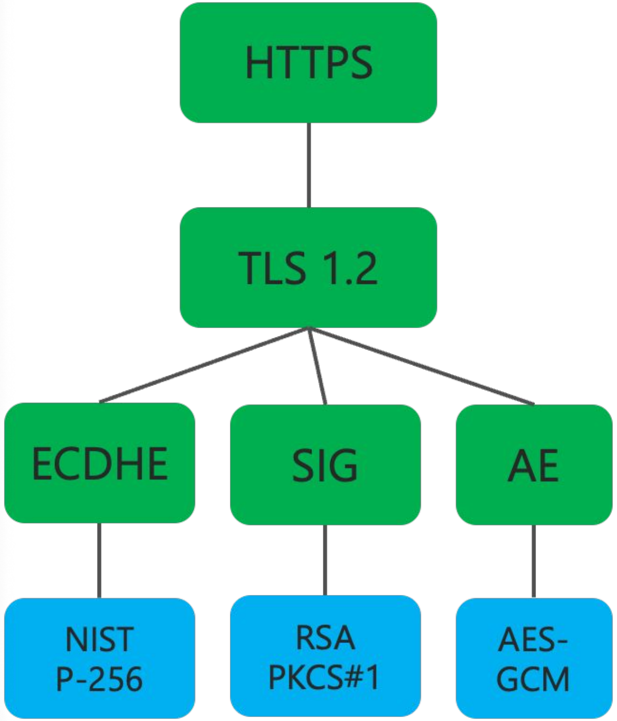
VSTTE 2024, Prague

CRYSPEN

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



crypto protocol



crypto algorithm

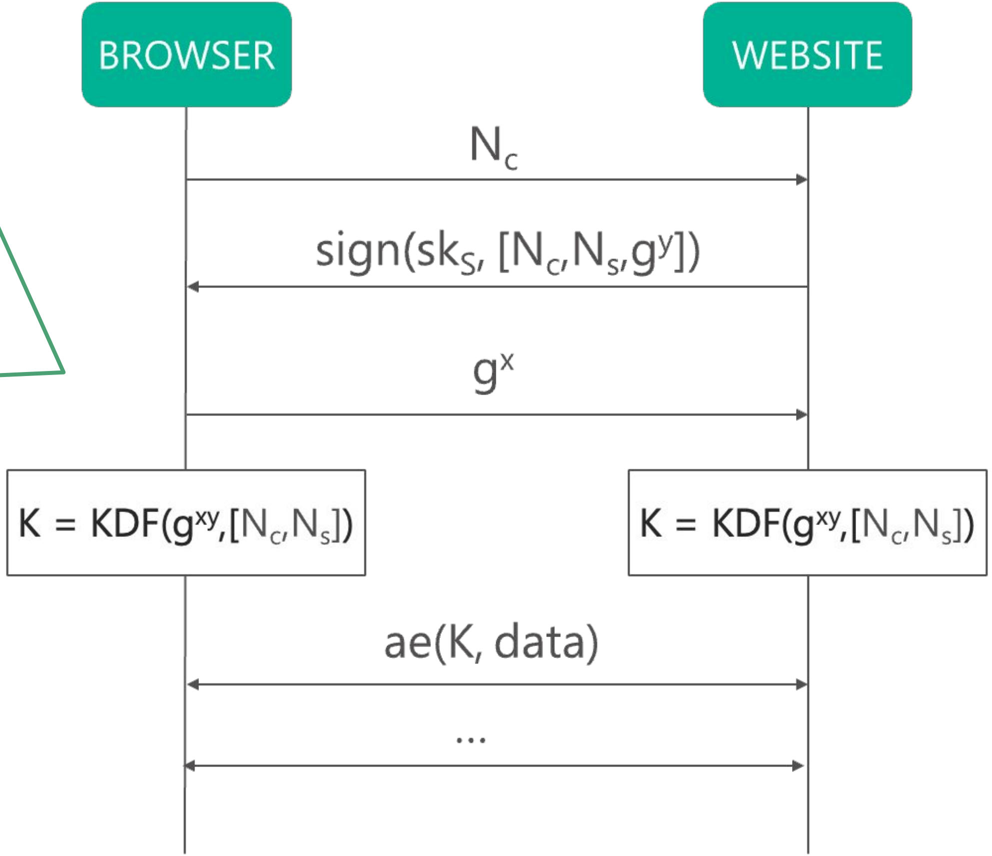
Creating Secure Channels (SSL 3.0-TLS 1.2)

Authenticated Key Exchange

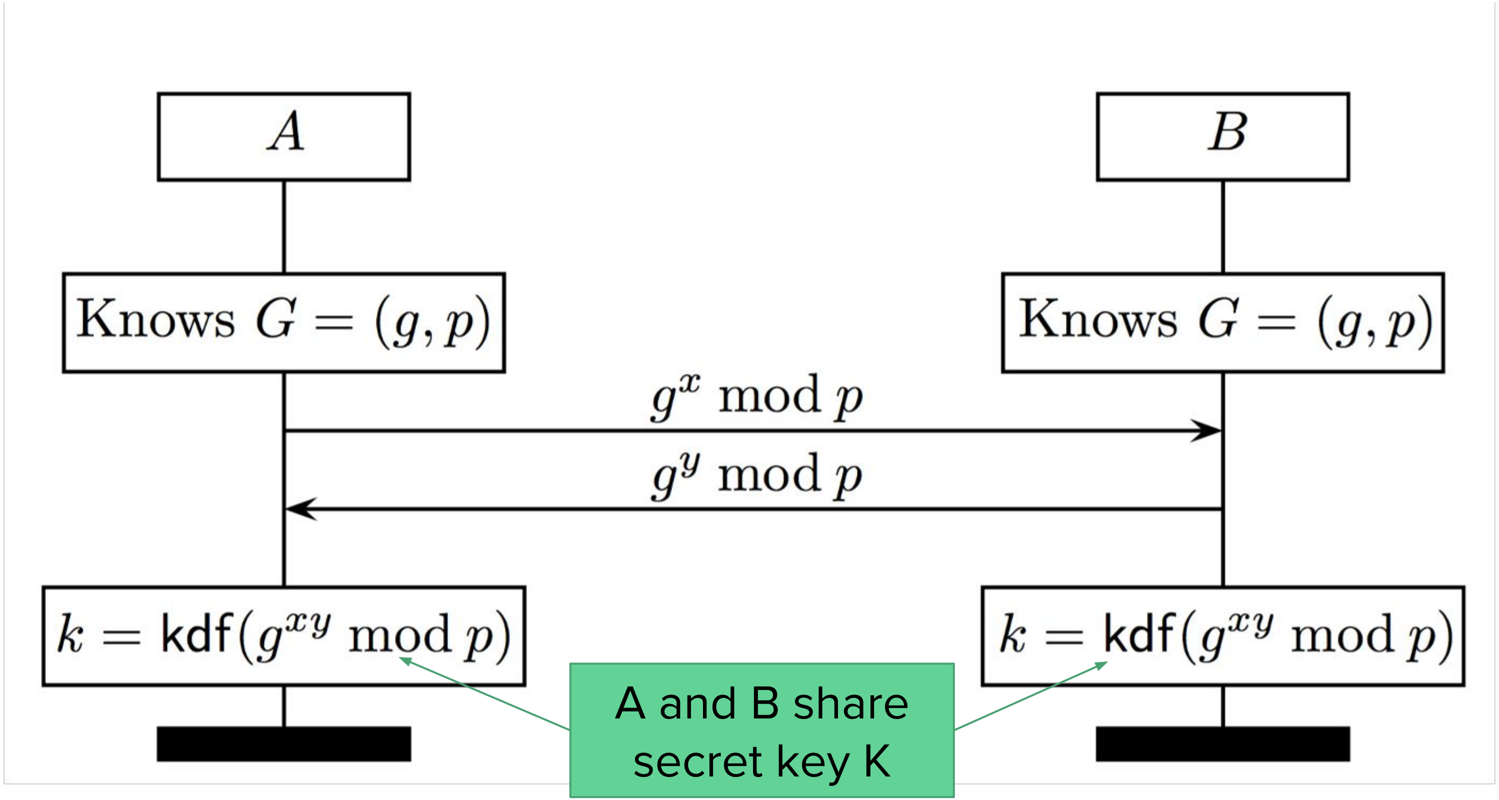
+

Authenticated Encryption

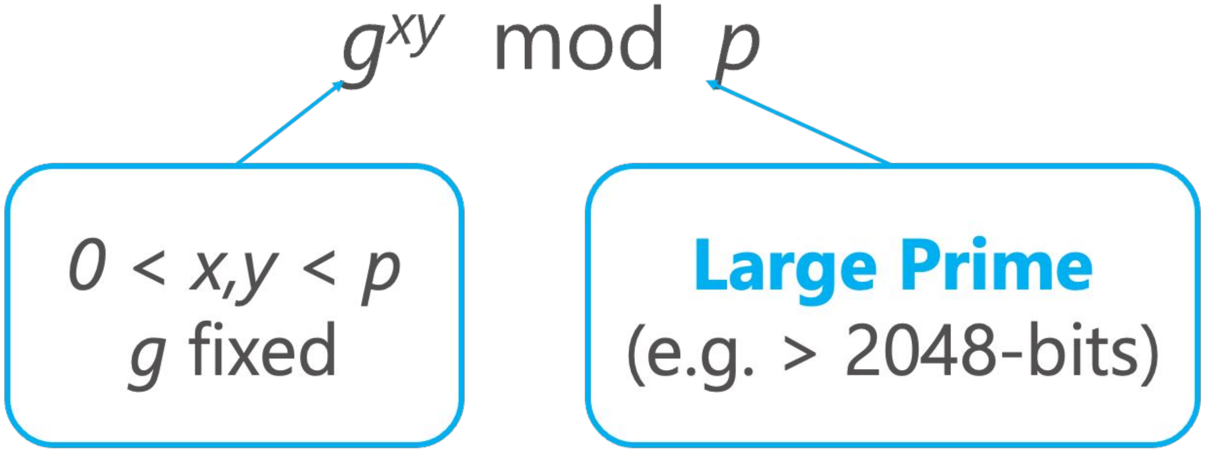
- Classic protocol design
- Many security proofs
- See e.g. SIGMA [2003]
- Solved Problem?



Diffie-Hellman Key Exchange (1976)



Diffie-Hellman Security Guarantee



The security of all DH-based protocols relies on a hardness assumption:
An attacker who does not know x or y cannot compute g^{xy} mod p

What can go wrong?

Bad Crypto: Weak Diffie-Hellman Groups

If the prime *p* is too small,
an attacker can compute the discrete log:
$y = \log(g^y \bmod p)$

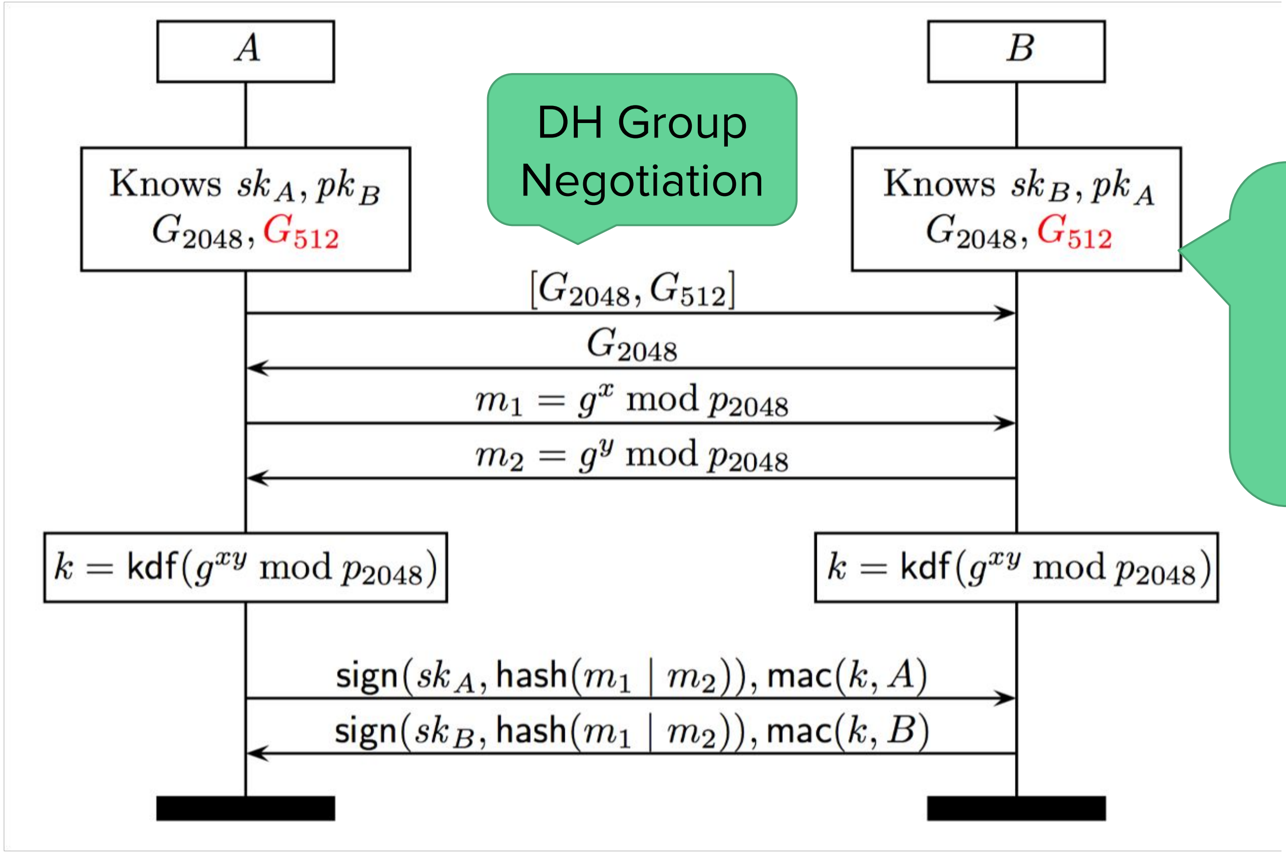
and hence compute the session key: $g^{xy} \bmod p$

Current discrete log computation records:

- [Joux et al. 2005] **431-bit prime**
- [Kleinjung et al. 2007] **530-bit prime**
- [Bouvier et al. 2014] **596-bit prime**
- [Boudot et al. 2019] **795-bit prime**

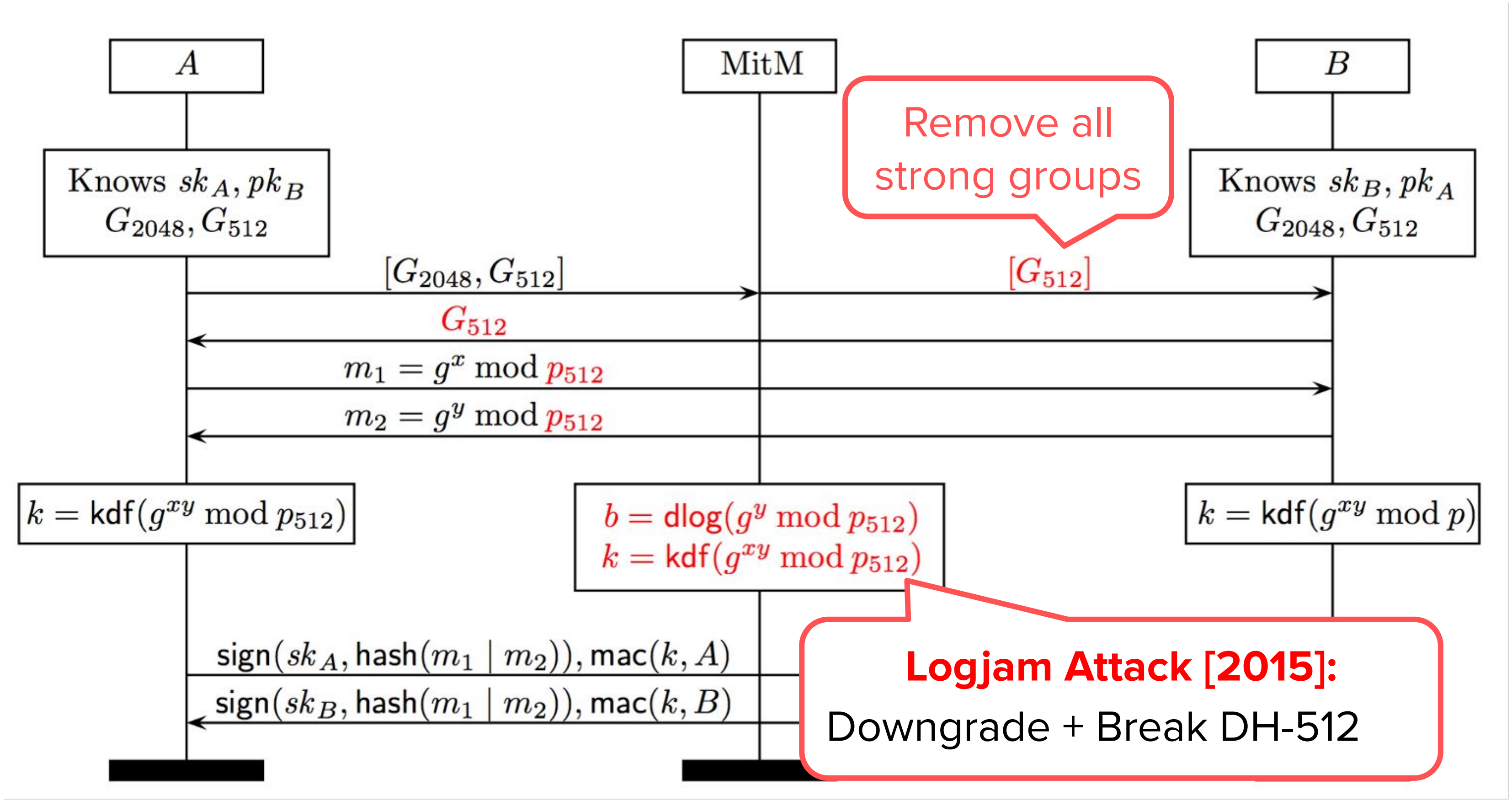
Broken (efficiently computable) by a Quantum Computer [Shor, 1994]

Protocol Flaw: Insecure Negotiation



Why support weak crypto?
backwards compatibility, ...

Protocol Flaw: Downgrade Attack



Coding Bugs: Cryptographic Computations

$$g^{xy} \mod p$$

$0 < x,y < p$

g fixed

Large Prime

(e.g. > 2048-bits)

Modular Exponentiation, implemented using bignum multiplication

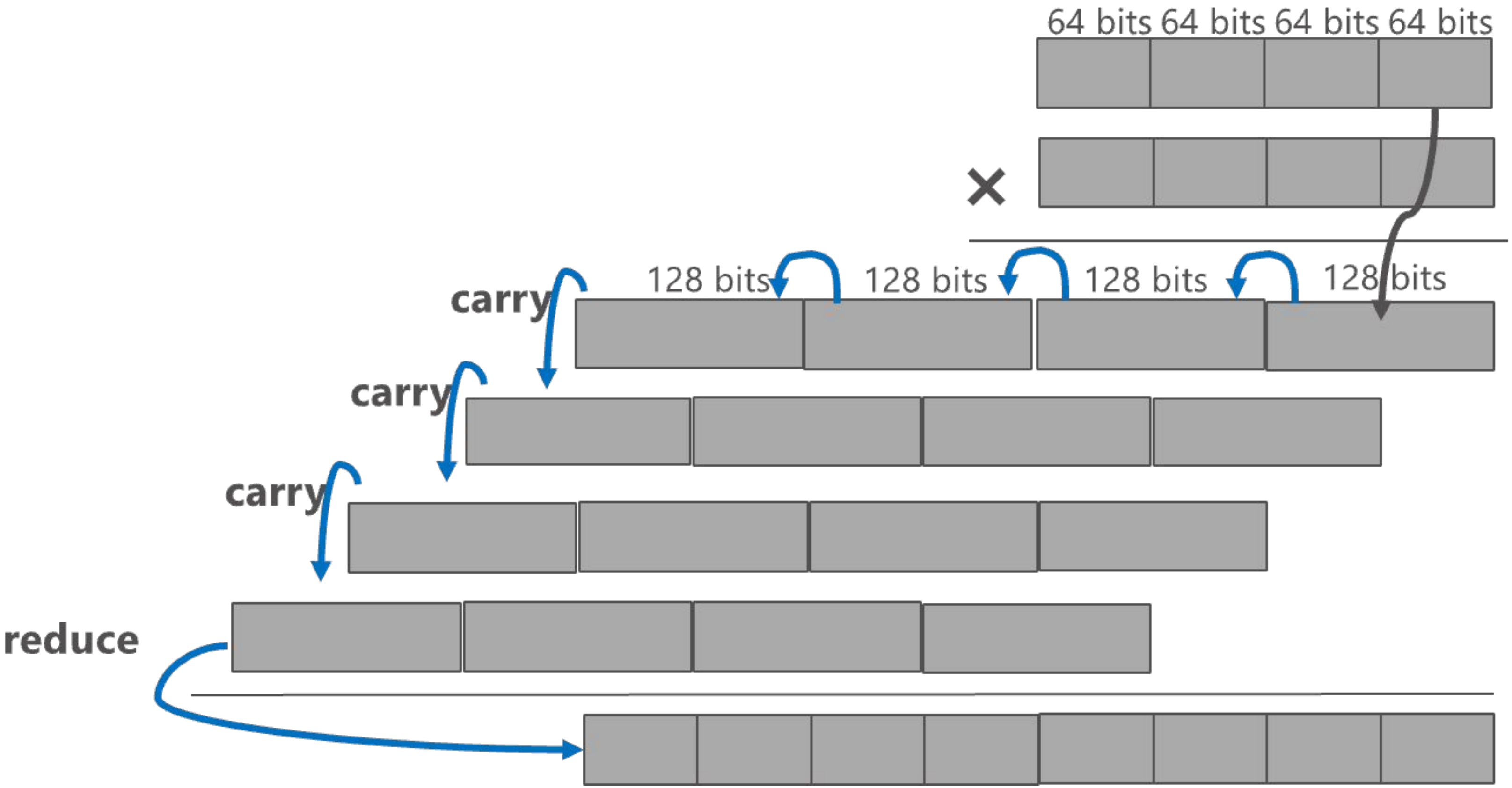
Can sometimes be the most expensive computation on a Web server

Coding Bugs: Textbook Multiplication

$$
\begin{array}{r}
13 \\
\times\ 10 \\
=\ 130
\end{array}
$$

```
      1101
  ×   1010
      ----
      0000
     1101
    0000
+  1101
 ---------
 10000010
```

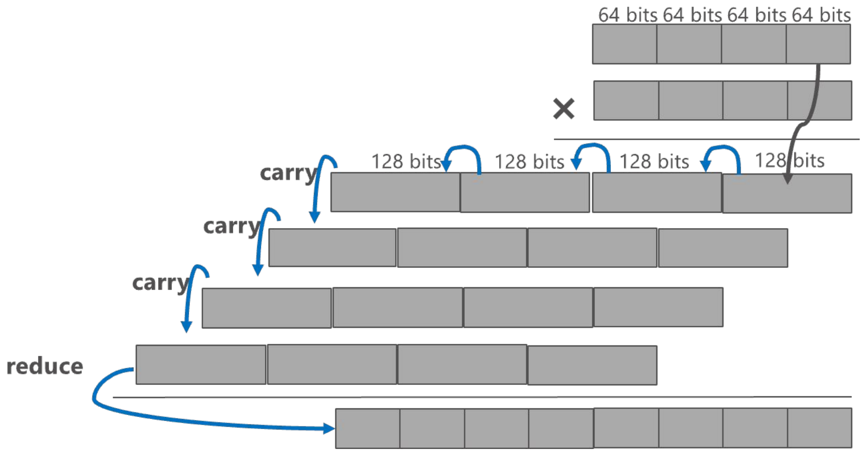
Coding Bugs: 256-bit modular multiplication



Coding Bugs: 256-bit modular multiplication

What can go wrong?

- Integer overflow
  (undefined output)
- Buffer overflow/underflow
  (memory error)
- Missing carry steps
  (wrong answer)
- Side-channel Attack
  (leaks secrets)



Coding Bugs: Side-channel attacks

```
        1101
  ×     1010
        ----
        0000
       1101
      0000
+    1101
   --------
   1000010
```

```
           1101
    ×      1010
           ----
          1101
+    1101
     --------
     1000010
```

Skipping 0s is faster!

- Fewer additions, carries

  ... but leaks information

- Runtime proportional to number of 1s in 1010
- Attacker can observe runtime to guess input
- May leak secret key!

Other Coding Bugs: Protocol Code

- Incorrect use of crypto primitives
  - Nonce reuse, public key validation, ...
- Parsing cryptographic formats
  - Ambiguities, incorrect parsing,, memory errors, ...
- Protocol state machine flaws
  - Authentication bypass, skip crypto operations, ...
- Crash or panic
  - Unexpected messages, memory leaks, ...

Formal methods can help!

Verified crypto protocol designs

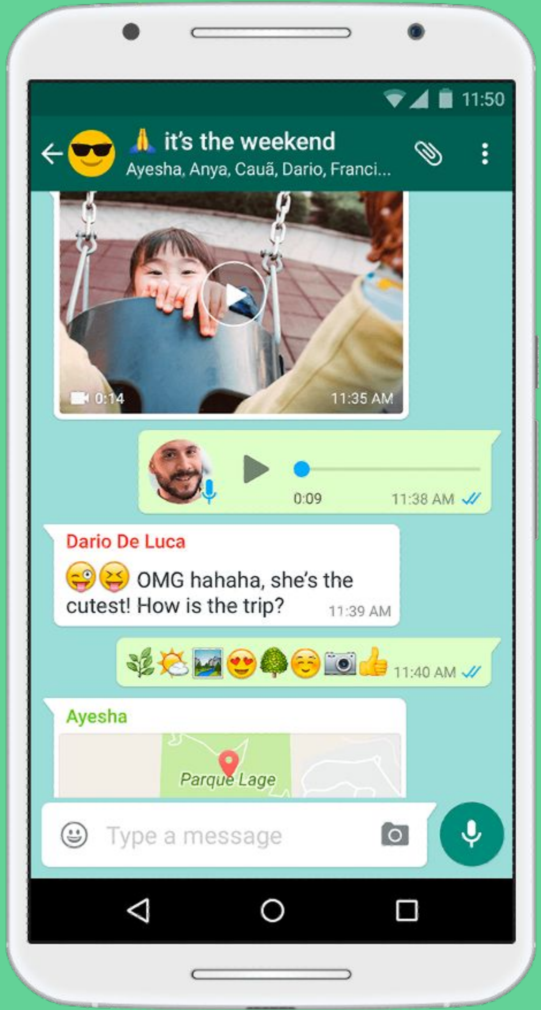
- Symbolic security analysis [ProVerif, Tamarin, DY*]
- Cryptographic proofs of security [CryptoVerif, EasyCrypt, Squirrel]

Verified crypto software

- Verified crypto libraries [F*, Coq, Isabelle, SAW, ...]
- Verified protocol code [F*, Dafny, Verus, ...]

A new opportunity:
many applications are now being updated to provide **Post-Quantum** security.

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



Analysing and Fixing Post-Quantum Signal

The (Classical) Signal Protocol

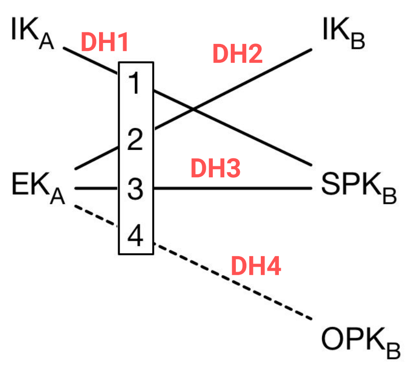
Modular design:

- **X3DH** handshake
- **Double Ratchet** for continuous key agreement

Important security guarantees:

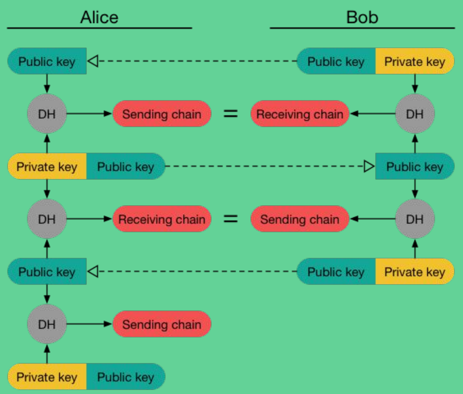
- Mutual authentication
- Post-compromise security
- Forward secrecy
- Deniability

X3DH



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

Double Ratchet



The (Classical) Signal Protocol

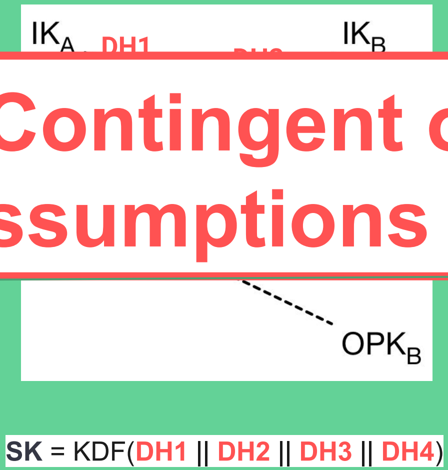
Modular design:

- X3DH handshake
- Double Ratchet for continuous key agreeme

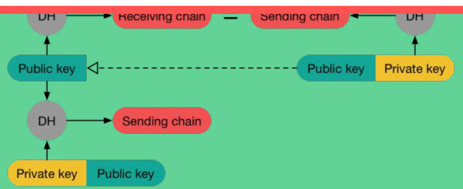
Important security guarantee

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

X3DH



Double Ratchet

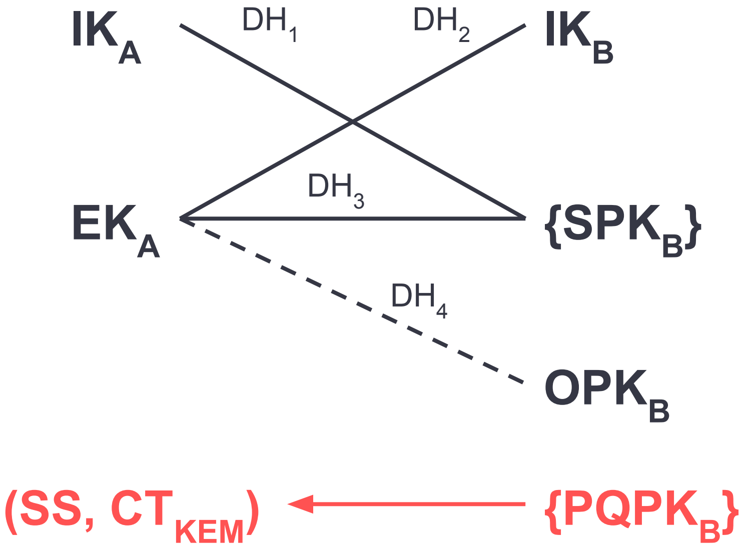


Contingent on Diffie-Hellman assumptions - quantum fragile!

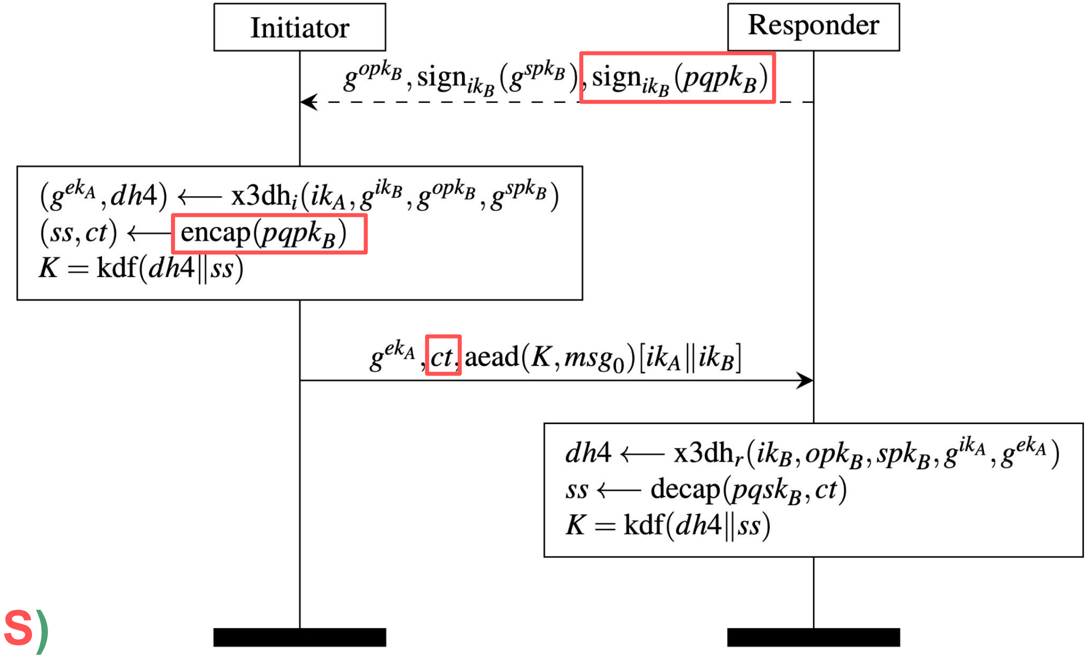
Harvest Now, Decrypt Later (HNDL) attacks:

Messages sent today
are vulnerable to
quantum attackers tomorrow

PQXDH Design: Add a PQ-KEM to X3DH



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

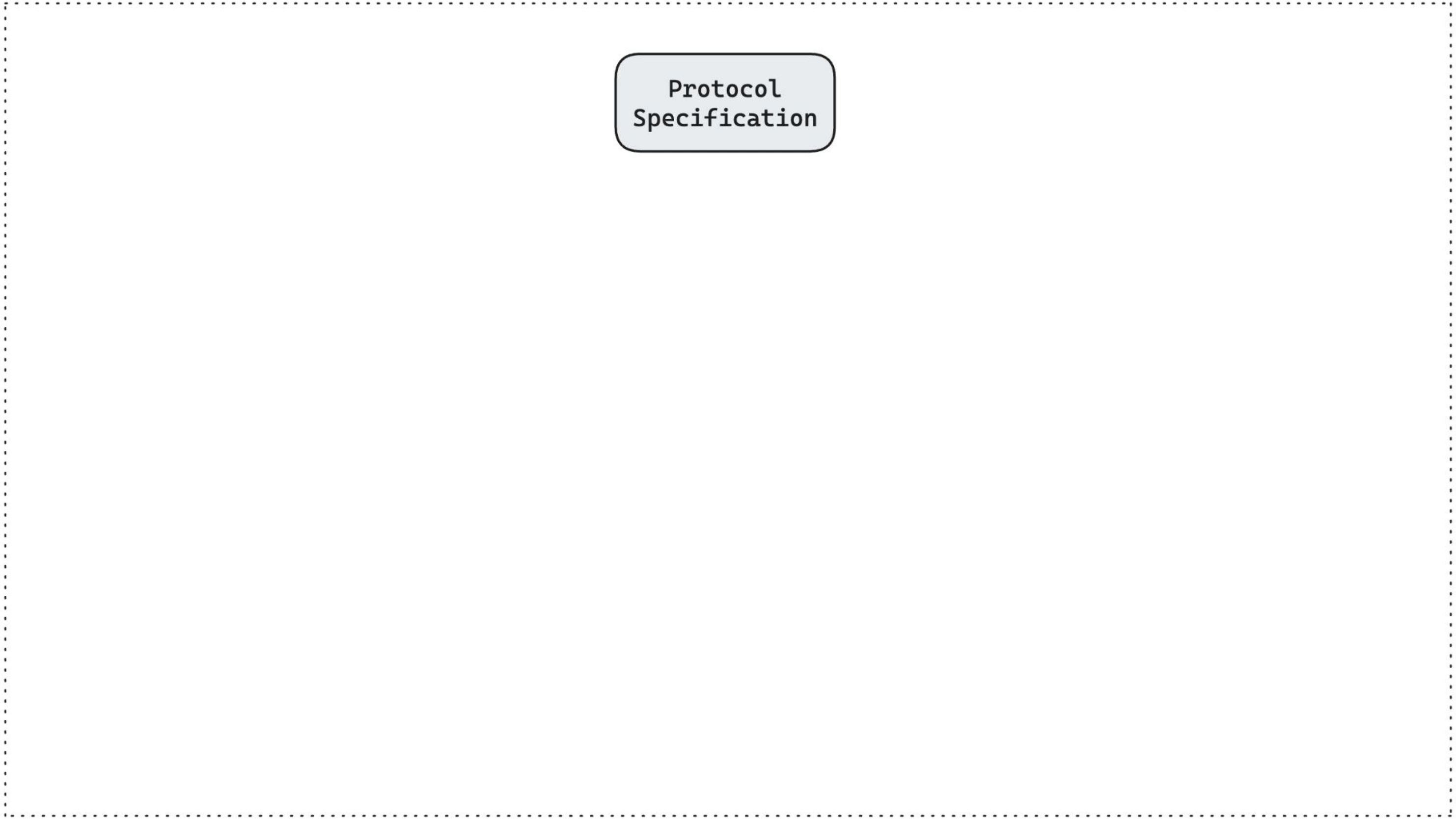


Analyzing PQXDH

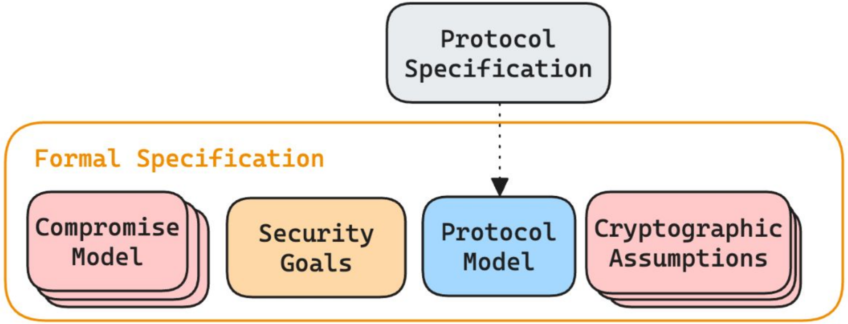
- PQXDH is a very small addition to X3DH.
- X3DH has been comprehensively analyzed in a variety of security models
  - Mutual Authentication, Confidentiality,Forward Secrecy

- Is PQXDH as secure as X3DH?
- Is it secure against an HNDL quantumadversary?

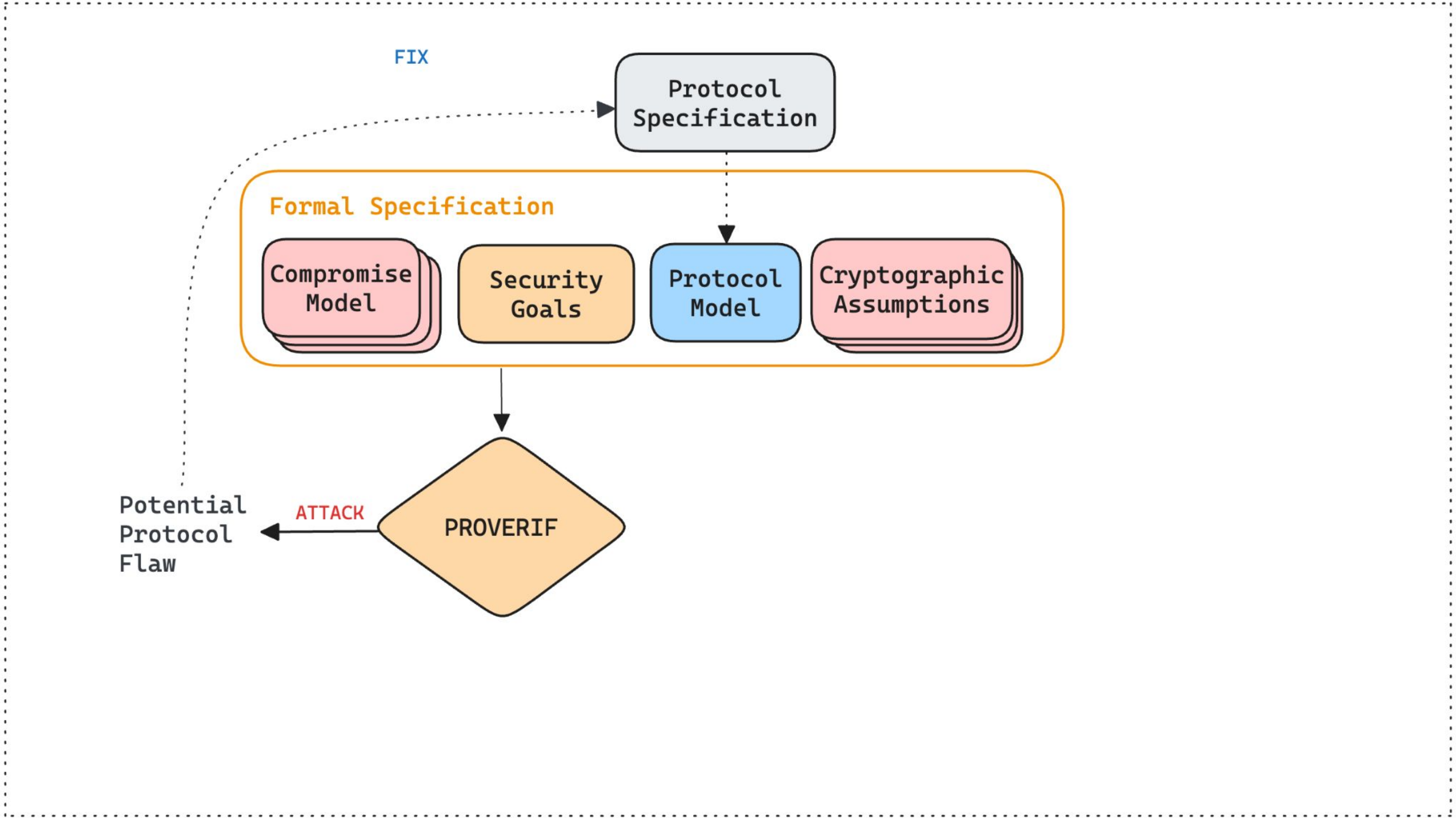
Our Formal Verification Methodology



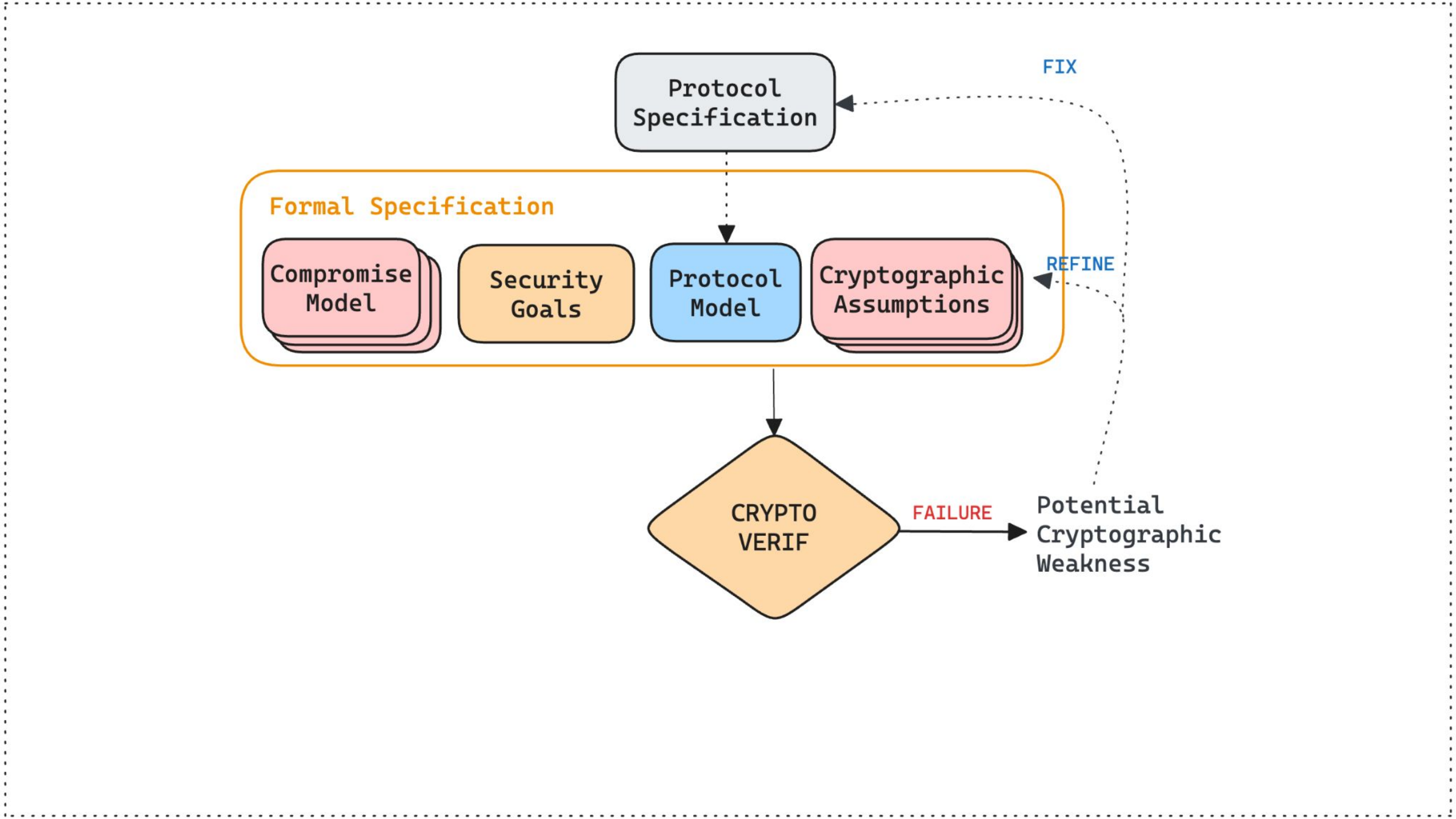
Our Formal Verification Methodology



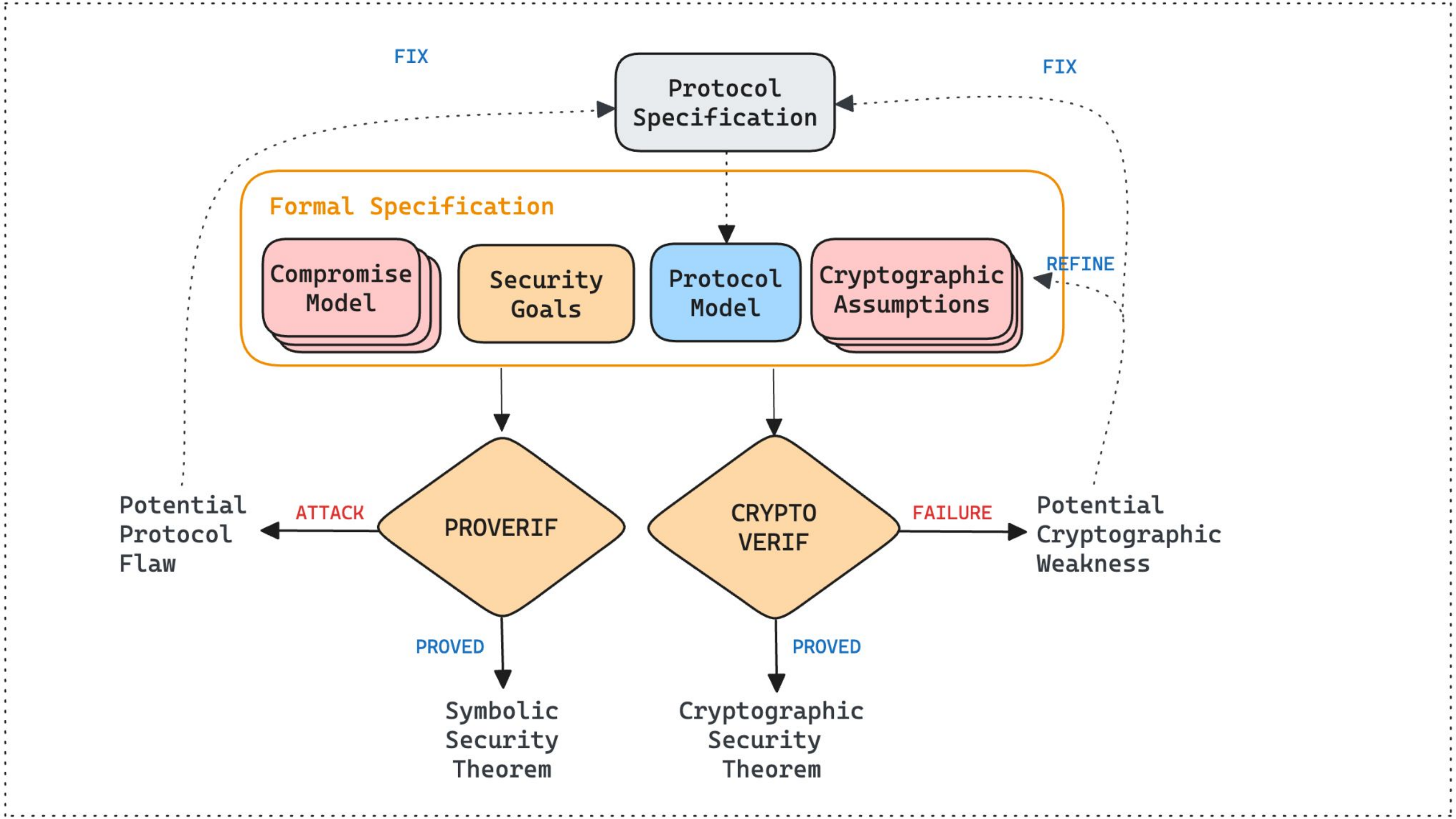
Our Formal Verification Methodology



Our Formal Verification Methodology



Our Formal Verification Methodology



Formally Specifying PQXDH

Single Message between Two Roles

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

Specification in Applied Pi Calculus

- Makes all computations precise.
- What is sent on the wire?
- What key encoding do we use?
- What exactly is signed/encrypted?
- How are all the keys derived?

```
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

Security goals as queries

- Secrecy, Authentication: trace properties
- Indistinguishability, Privacy: equivalence properties

Fully automated analysis

- Finds attacks and produces traces
- No attack found ⇒ symbolic security theorem
- Might not terminate!

```
(* 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))).
…

Game-Based Security Proofs with CryptoVerif

Computational crypto model

- Standard cryptographic assumptions
- User-defined assumptions as equivalences
- Probabilistic polynomial-time adversary

Proof: sequence of game transformations

- Requires some manual guidance
- Machine-checked transformations
- Computes concrete advantage formulas
- Proof failure may indicate attack, no trace

```
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
}
```

Modeling the Quantum Adversary

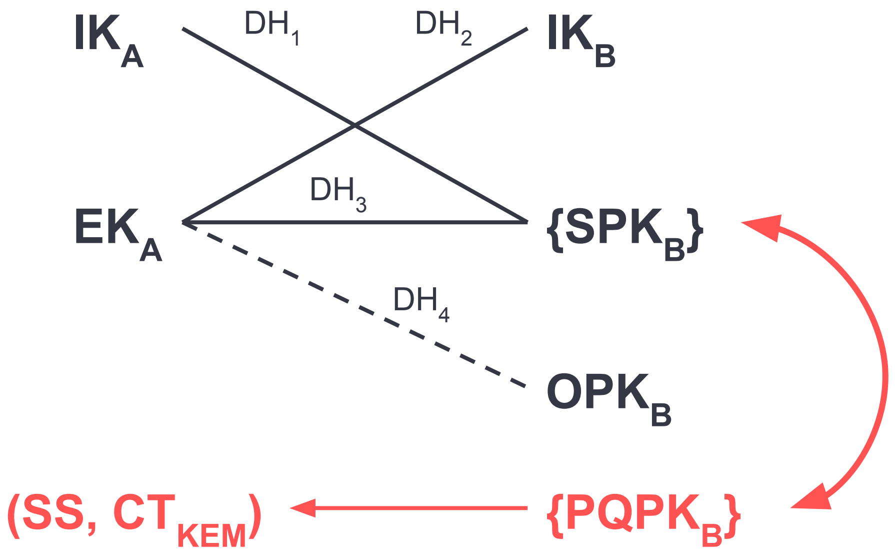
Passive Quantum Adversary Model (Harvest-Now-Decrypt-Later)

- Adversary can break DH *after* the session is over
- PQ primitives (e.g. PQ-KEM) remain secure

Symbolic and Computational Analysis

- ProVerif automatically searches for attacks that rely on broken primitives
- CryptoVerif checks that the classical game-based proof still holds against passive quantum attackers
  - *Post-quantum sound CryptoVerif and verification of hybrid TLS and SSH key-exchanges*, Blanchet, Jacomme, IEEE CSF 2024

Key Confusion Attack on PQXDH



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

Attacker swaps keys and signatures to break PQ security of PQXDH

ProVerif finds this attack if:

- **the key encodings can collide, and**
- **public keys are not validated**

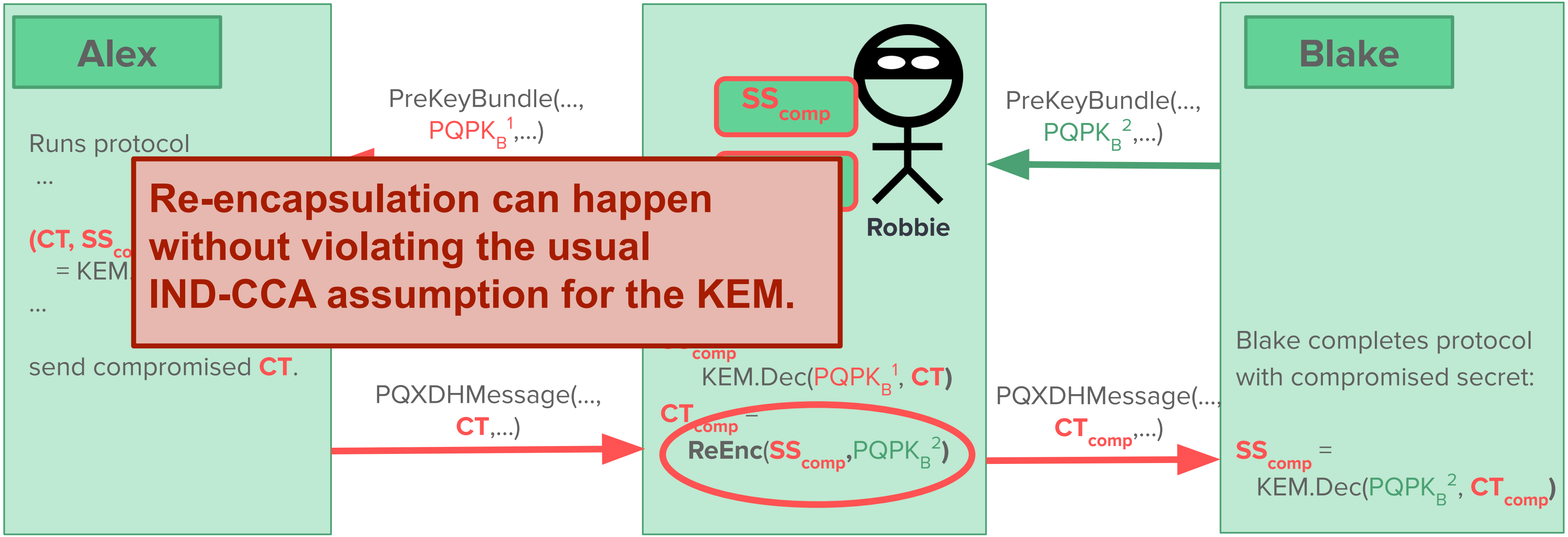
This is representative of a general class of **cross-protocol attacks** between old and new versions of the same protocol.

Easy Fix: Ensure all key/message/signature encodings have disjoint co-domains.

Signal implementation already does this

KEM Re-encapsulation Vulnerability

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



PQXDH Revision and Security Theorems

These findings led to a new revision of the PQXDH protocol:

- We required **AEAD** to be post-quantum **IND-CPA** and **INT-CTXT**
- Restricted the ranges of encodings to be disjoint
- Added **$PQPK_B$** to AD when it isn't already bound within the KEM

With these changes we can prove that PQXDH meets its classical and PQ security requirements in the symbolic, computational, and HNDL quantum models.

The full process: analysis, fix, proof, new spec took 1 calendar month.

But is the Signal *Implementation* Secure?

Is the new PQ crypto code PQXDH relies on implemented correctly?

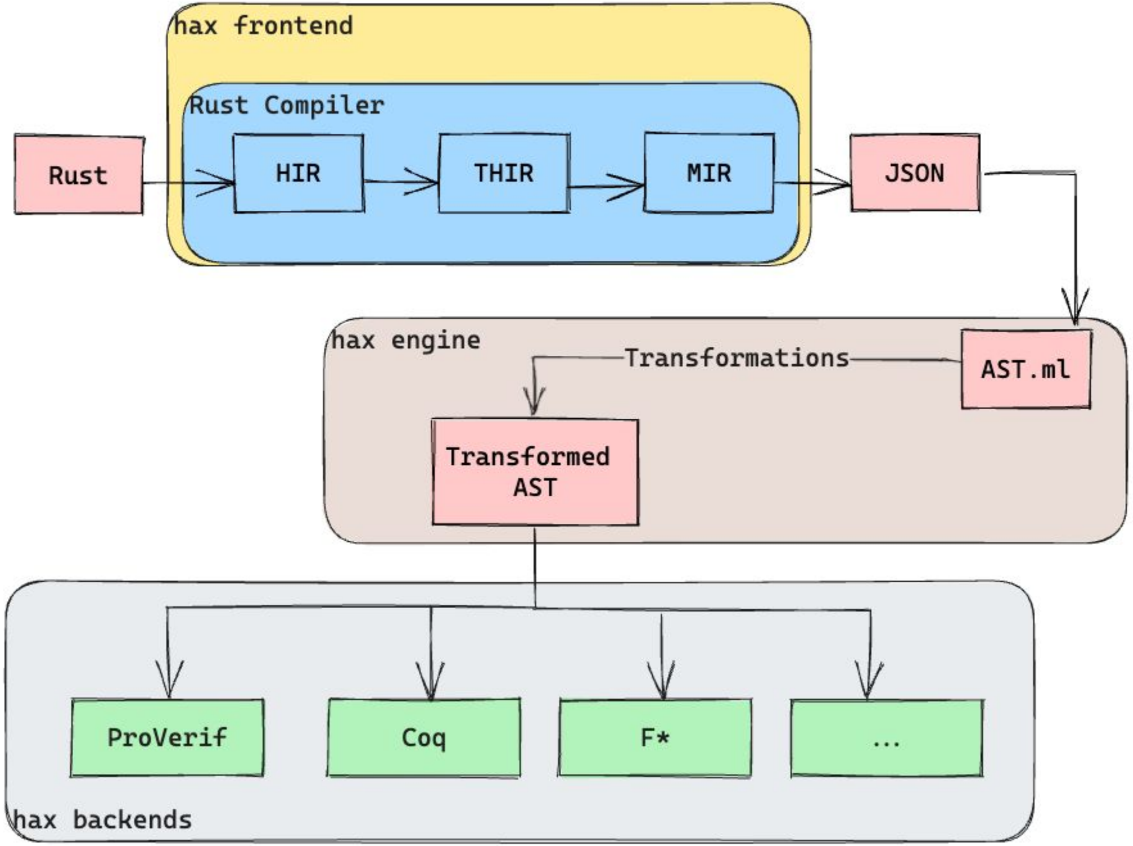
FIPS 203 (Draft)

Federal Information Processing Standards Publication

Module-Lattice-based Key-Encapsulation Mechanism Standard

Category: Computer Security **Subcategory: Cryptography**

hax: linking Rust code with proof backends

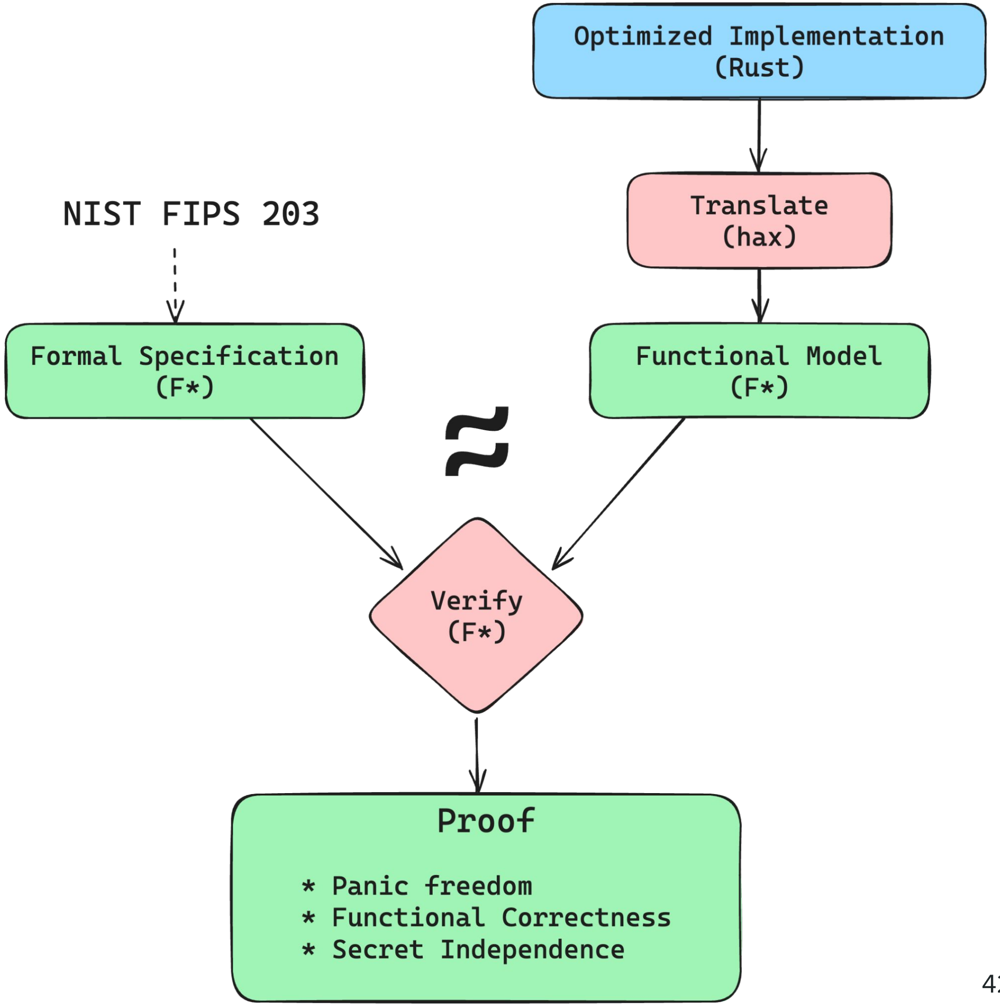


CRYSPEN

AARHUS UNIVERSITY

Inria

Verifying crypto code written in Rust using hax and F*



Writing Crypto Code in Rust

```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, so cannot directly use modulus)

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 % 3329 ≈ input % 3329`
`&& -3329 < result < 3329`

Enforcing Secret Independence

Type-based 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

Prevents a large class of remote timing attacks (at source level).

Does not prevent compiler-induced leaks, micro-architectural attacks, ...

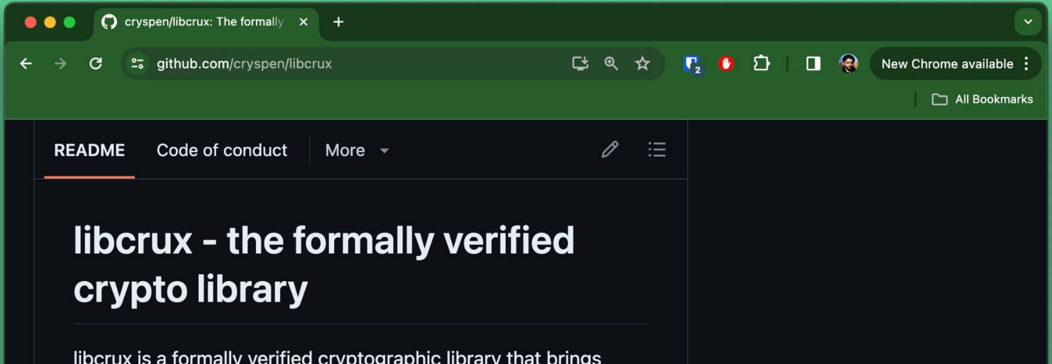
KyberSlash: a new timing vulnerability

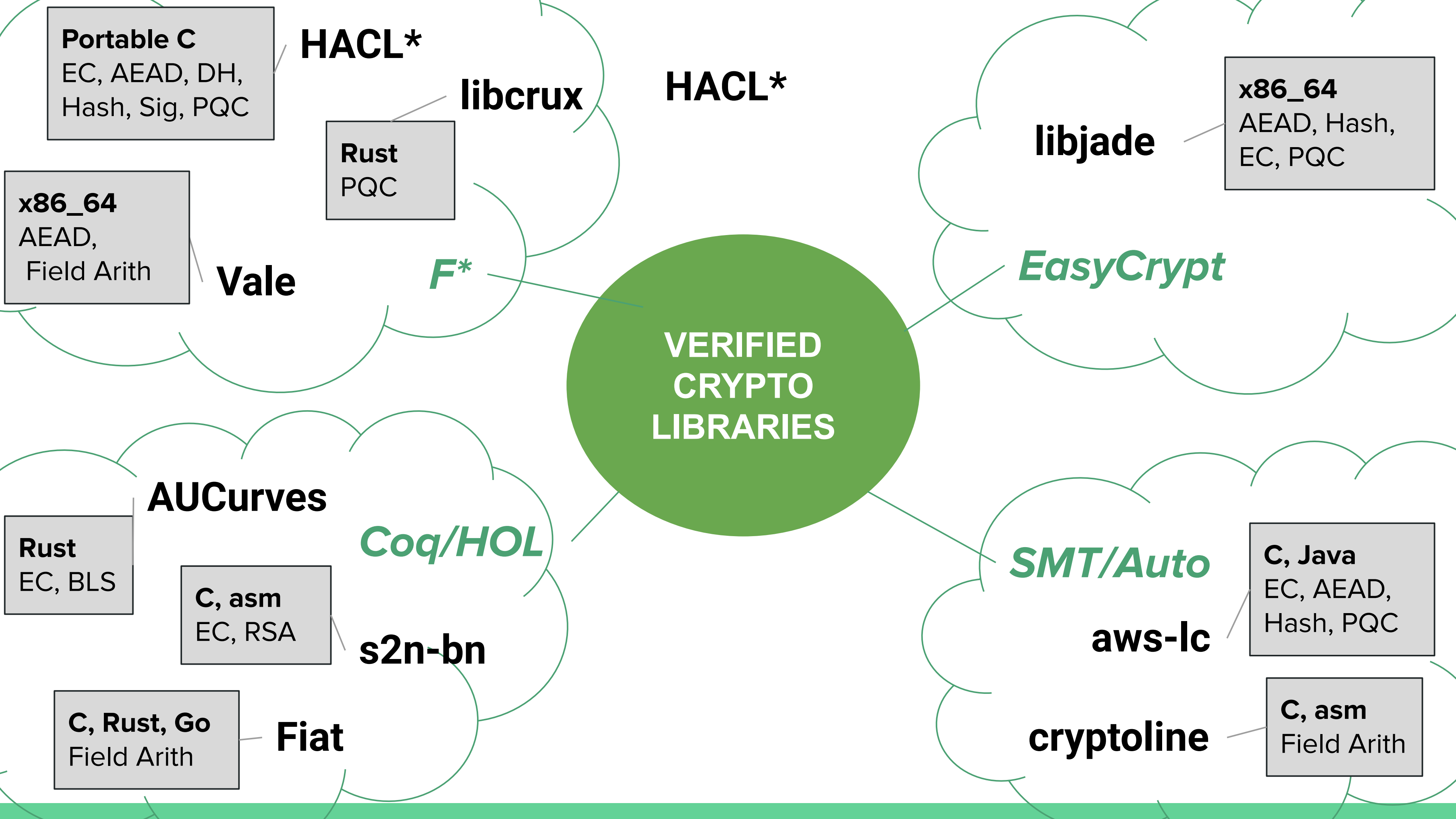
```
void poly_tomsg(uint8_t msg[KYBER_INDCPA_MSGBYTES], const poly *a)
{
  unsigned int i,j;
  uint16_t t;

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

Bug found during Formal Verification of our Rust code!

We built an **optimized**, **portable**, formally **verified** implementation of ML-KEM in Rust and C that is now deployed in Firefox.




VERIFIED CRYPTO LIBRARIES
F*
HACL*
Portable C
EC, AEAD, DH, Hash, Sig, PQC
libcrux
Rust
PQC
HACL*
Vale
x86_64
AEAD, Field Arith
EasyCrypt
libjade
x86_64
AEAD, Hash, EC, PQC
Coq/HOL
AUCurves
Rust
EC, BLS
s2n-bn
C, asm
EC, RSA
Fiat
C, Rust, Go
Field Arith
SMT/Auto
aws-lc
C, Java
EC, AEAD, Hash, PQC
cryptoline
C, asm
Field Arith

Challenges and Research Directions

Modeling and verifying security against active quantum adversaries

- Moving beyond HNDL, handling post-quantum signatures

Verifying cryptographic protocol implementations

- Challenging for automation, ongoing work on TLS, MLS, Signal, ...

Verifying privacy-preserving crypto mechanisms and protocols

- Zero-Knowledge proofs, Fully Homomorphic Encryption, MPC, etc.

Applying formal methods to larger cryptographic applications

- Build tools usable by developers, applicable to Rust, Go, C, ...

Conclusions

- Just switching to brand new crypto does not improve security
  - We may be introducing new attacks that did not exist before
- Formal methods can help answer questions about crypto artifacts
  - We still need to ask the right questions from multiple angles
  - Systematic tool-based analyses can help head off issues early
- Crypto is not static, so proofs and implementations also need to evolve
  - A need for proof engineering, maintenance, continuous integration
  - A need for custom, usable tools that crypto developers can use

Questions?

- *SoK: Computer-Aided Cryptography*
  [Barbosa, Barthe, Bhargavan, Blanchet, Cremers, Liao, Parno, IEEE S&P 2021]
- *Formal verification of the PQXDH Post-Quantum key agreement protocol for end-to-end secure messaging*
  [Bhargavan, Jacomme, Kiefer, Schmidt, Usenix Security 2024]
- libcrux: https://github.com/cryspen/libcrux
- hax: https://github.com/hacspec/hax

CRYSPEN

https://cryspen.com