

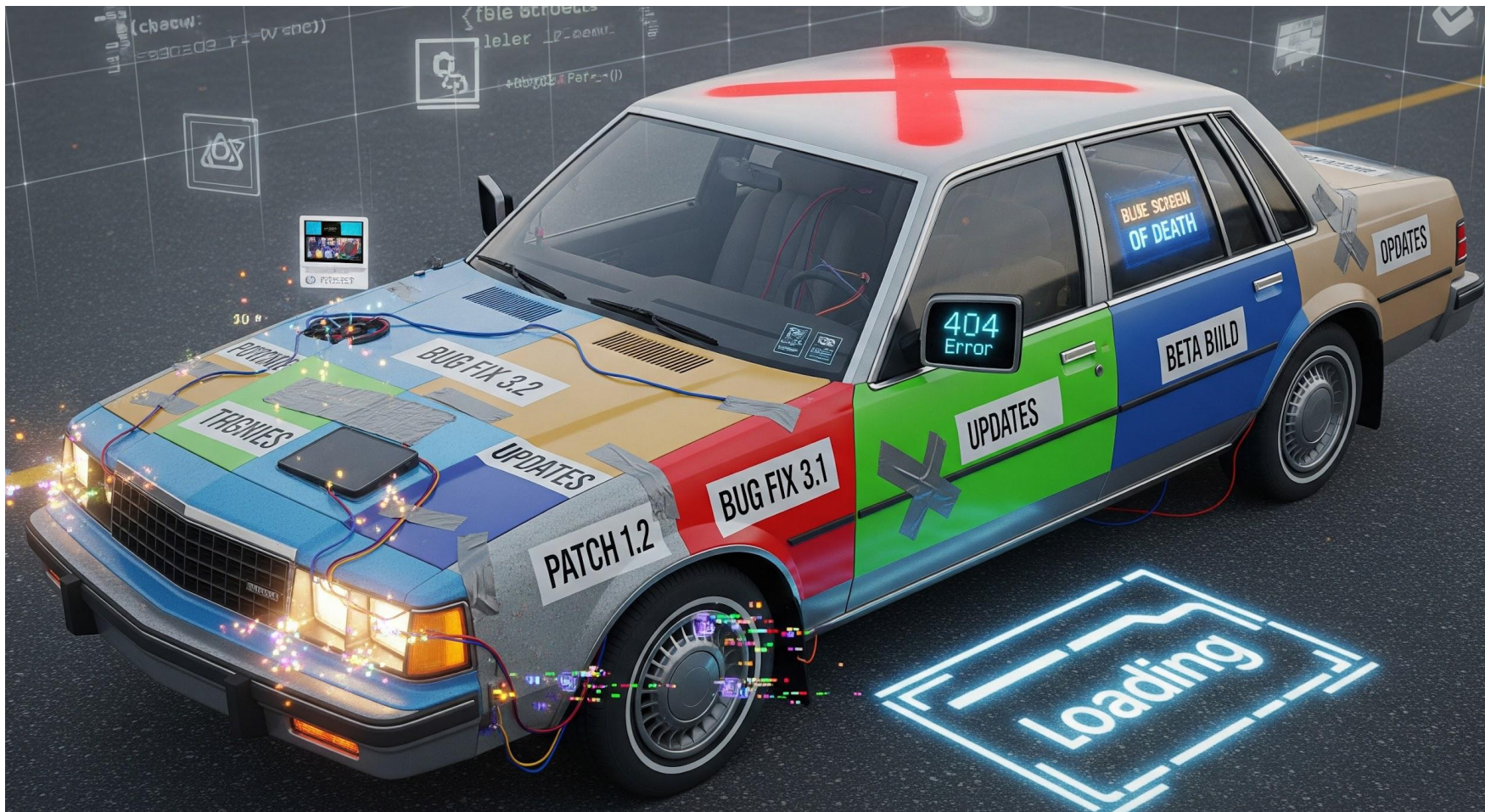
High-Assurance Post-Quantum Cryptography for Security-Critical Applications

Karthikeyan Bhargavan

Joint work with many others at Cryspen, Inria, Signal, ...

PQCSA Workshop, 2025





A car that's built like software, with all of the bugs and patching.

Crypto to the rescue?

Many software-driven features are protected using cryptography

- Secure Software Updates: Digital Signatures
- V2X, Intra-Vehicular Communication: Secure Channels
- Secure Charging, Keyless Entry: Mutual Authentication

Standardized and proprietary mechanisms and protocols

- The core of the trusted computing base of automotive software
- Long-lived designs and implementations
- Design flaws and Implementation bugs can be catastrophic

The Post-Quantum Transition presents new risks and dangers as well as new opportunities to do better for a new generation of cars!

Formal methods can
speed development and
clarify security of
Post-Quantum Crypto
algorithms and protocols
in automotive systems.

Formal Methods for Crypto

- Computer-Aided Cryptography, a.k.a. High Assurance Cryptography

“Applying formal, **machine-checkable** approaches to the design, analysis, and implementation of cryptography.”

SoK: Computer-Aided Cryptography, IEEE S&P 2021

Barbosa, Barthe, Bhargavan, Blanchet, Cremers, Liao, Parno

- Analyze cryptographic **designs** early to find attacks or uncover assumptions
- Comprehensively analyze **specifications** and standards before publication
- Formally verify efficient **implementations** to prevent bugs and side-channels
- and **repeat** these steps over and over again as these artifacts evolve

A Historical Example: TLS

LCL - Banque et assurance

lcl.fr

Incognito



Ma vie. Ma ville. Ma banque.

Vous êtes un particulier

Nous contacter

Devenir client

Mon espace

COMPTES ET CARTES

EPARGNER

EMPRUNTER

S'ASSURER

SIMULATEURS ET DEVIS

Ma Vie

Le Mag

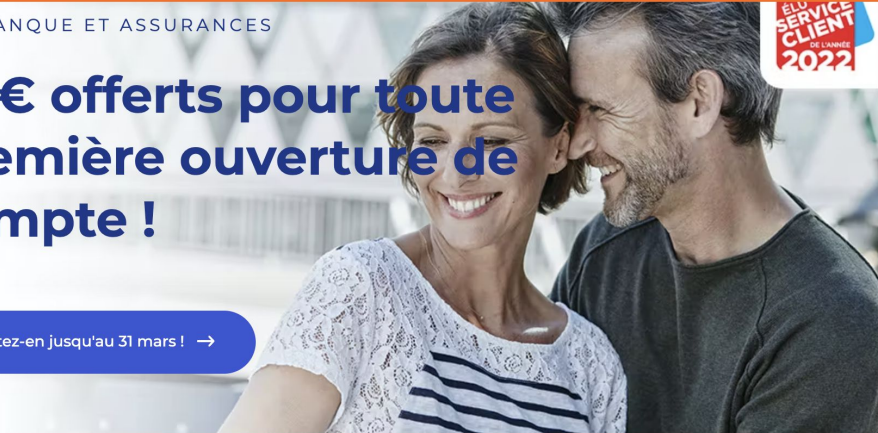
Découvrir LCL

ALERTE FRAUDE : Nous constatons une vague d'appels téléphoniques frauduleux, pour en savoir plus cliquez ici.

LCL BANQUE ET ASSURANCES

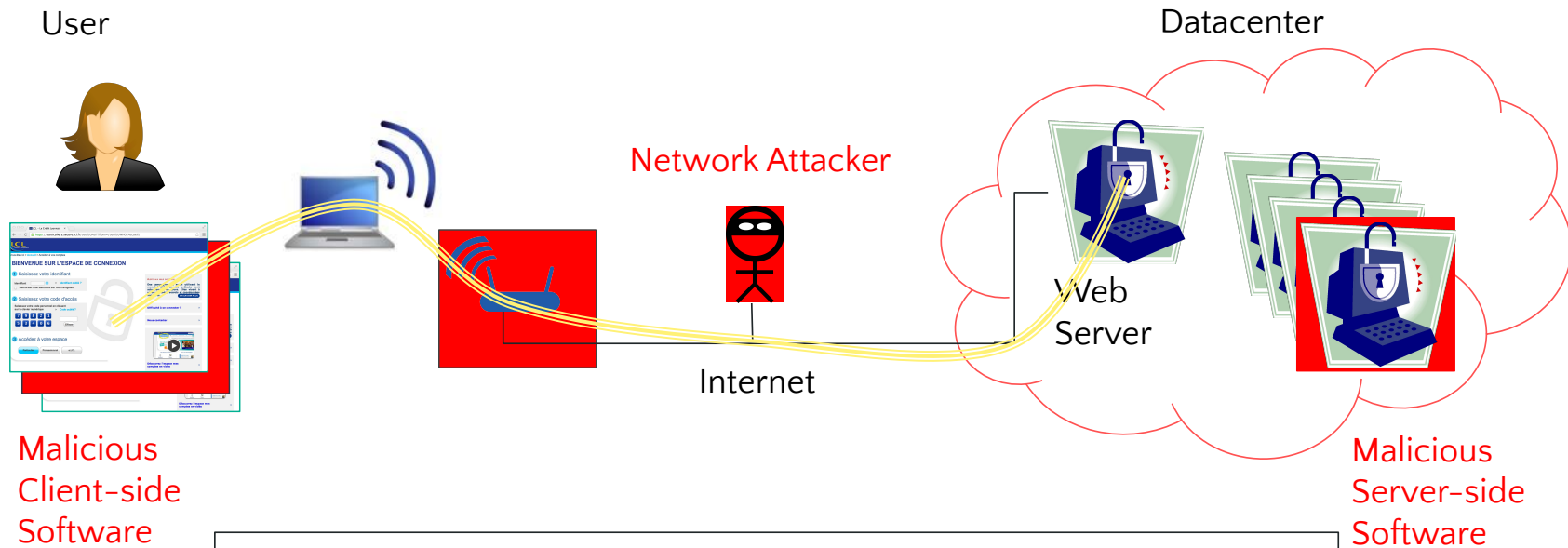
40€ offerts pour toute première ouverture de compte !

Profitez-en jusqu'au 31 mars ! →

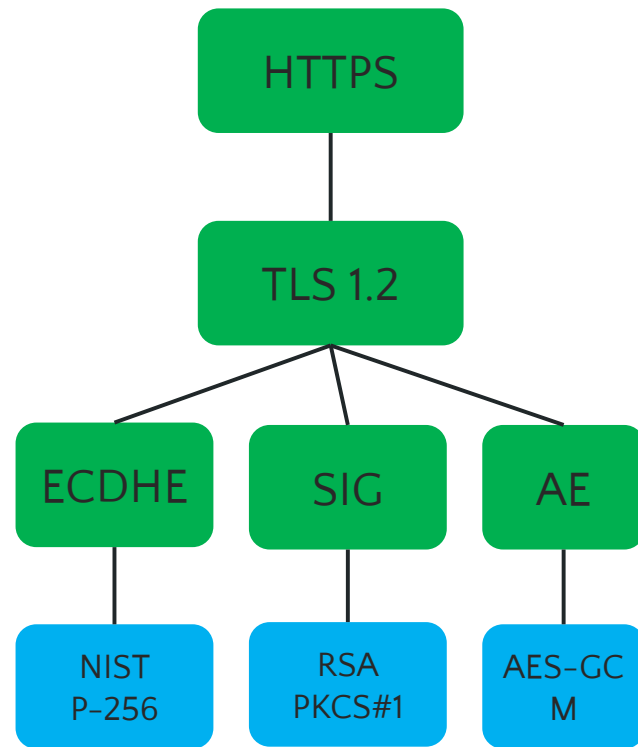
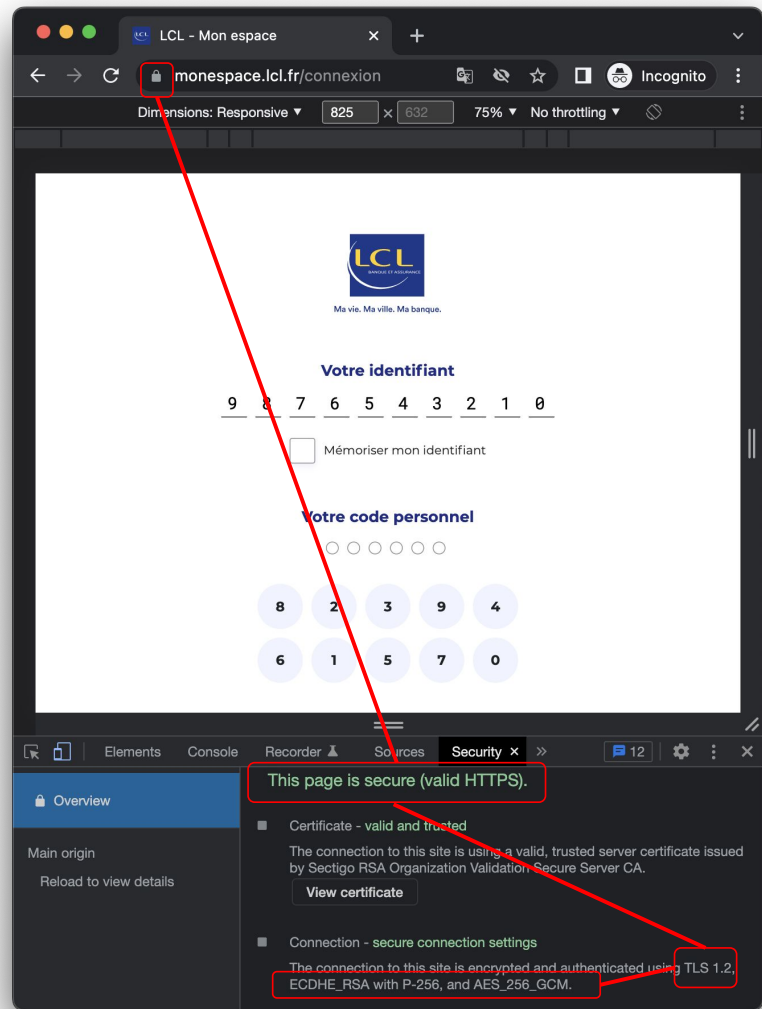


Parlons de vos besoins

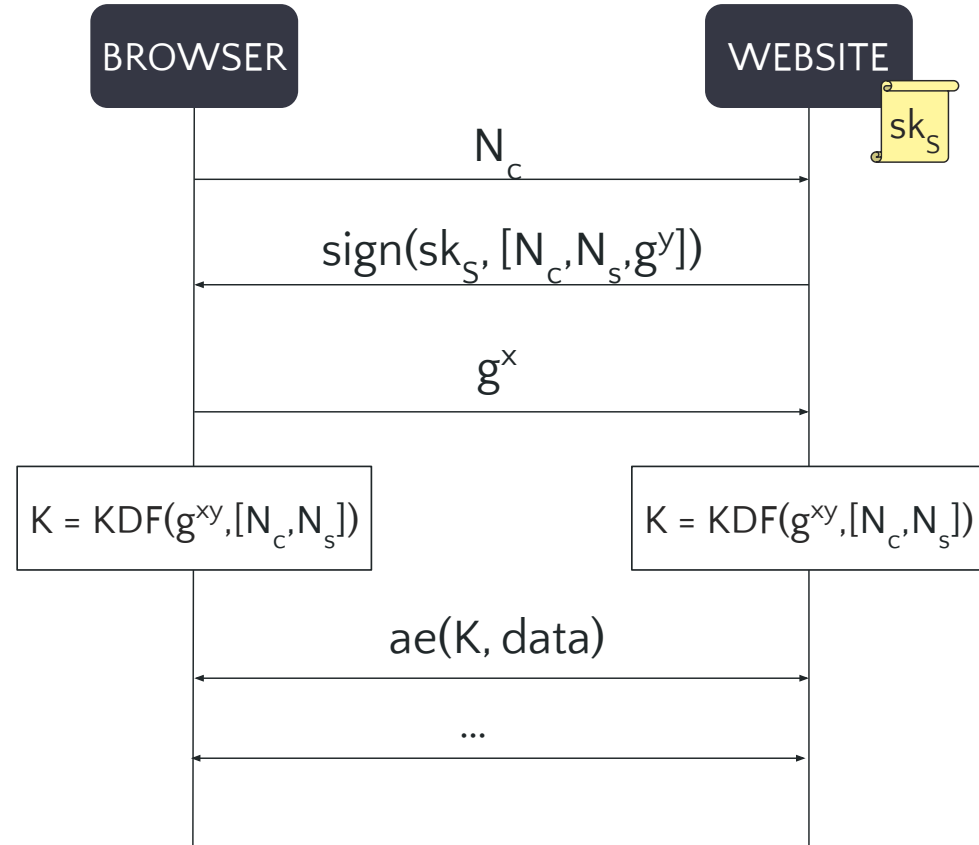
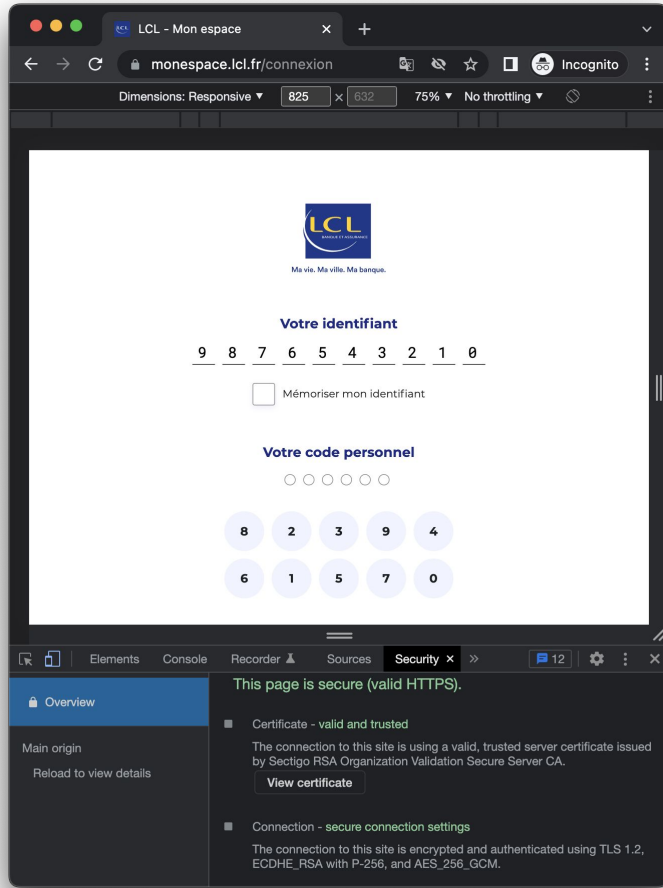
Secure Channels over Insecure Networks



- We need a secure (authentic, confidential) channel
- Security against powerful attackers who may
 - **Read** all data sent on the network
 - **Tamper** with the contents of messages
 - **Impersonate** a user or a server



Secure Channels in TLS [1994-2008]



Secure Channels in TLS [1994-2008]

Classic Two-Stage Protocol

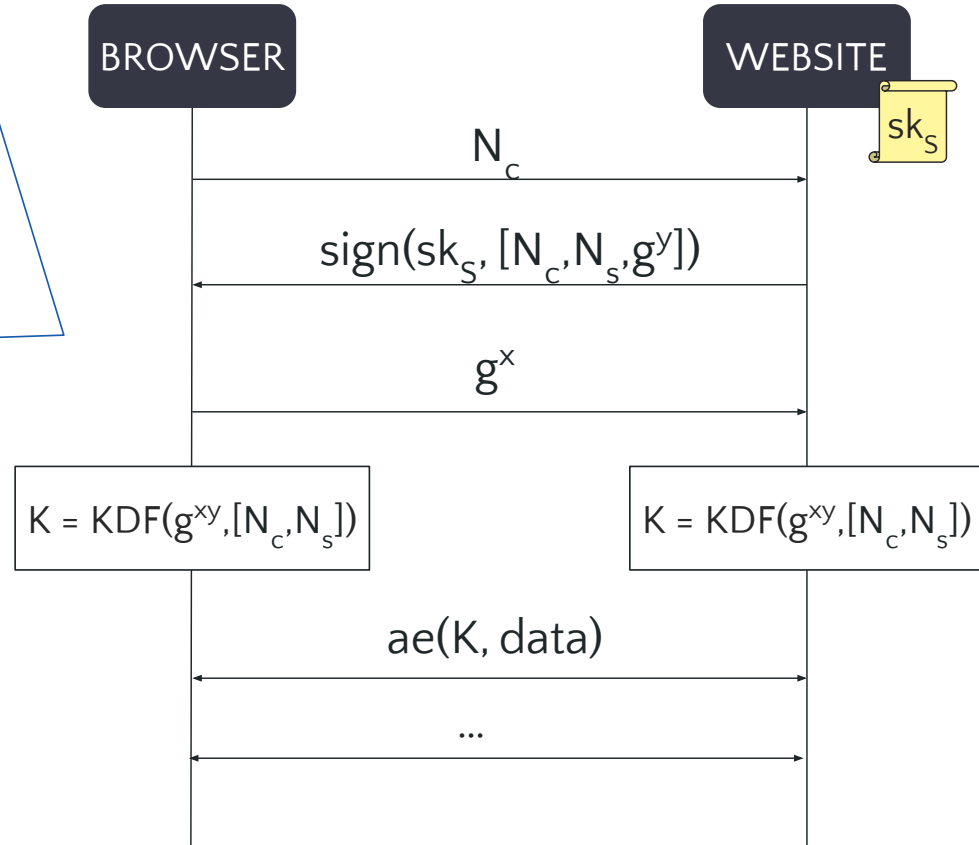
1. **Authenticated Key Exchange**

Both parties compute a
shared secret key K

2. **Authenticated Encryption**

Both parties exchange streams
of data encrypted under K

- Many variants (IPsec, SSH, TLS)
- Many security models & proofs
- So, is this a solved problem?



Many Many Practical Attacks on TLS [2011-16]

- **BEAST** CBC predictable IVs [Sep'11]
- **CRIME** Compression before Encryption [Sep'12]
- **RC4** Keystream biases [Mar'13]
- **Lucky 13** MAC-Encode-Encrypt CBC [May'13]
- **3Shake** Insecure resumption [Apr'14]
- **POODLE** SSLv3 MAC-Encode-Encrypt [Dec'14]
- **FREAK** Export-grade 512-bit RSA [Mar'15]
- **LOGJAM** Export-grade 512-bit DH [May'15]
- **SLOTH** RSA-MD5 signatures [Jan'16]
- **DROWN** SSLv2 RSA-PKCS#1v1.5 [Mar'16]
- **SWEET32** 3DES collisions [May'16]

What goes wrong in protocols like TLS?

Crypto Weaknesses

- RC4, 3DES, MD5, PKCS#1 v1.5

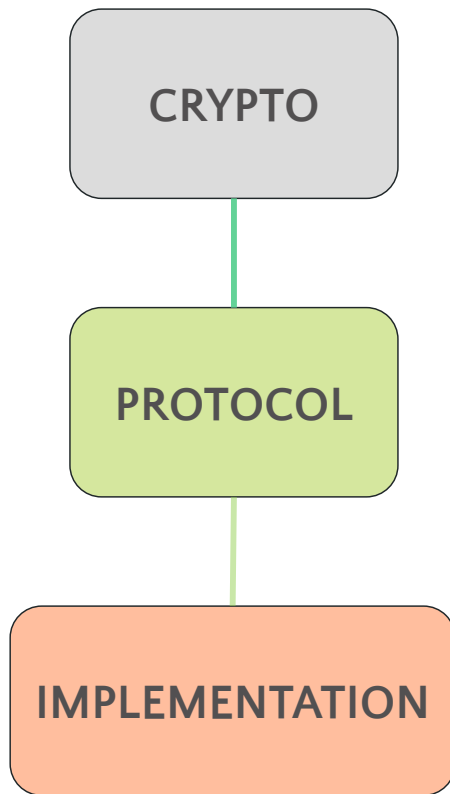
Protocol Design Flaws

- Downgrade attacks, Transcript collisions

Implementation Bugs

- State machine bugs, Memory safety

Often, a mix of all the above!



Lessons for Automotive Systems

All crypto can eventually be broken

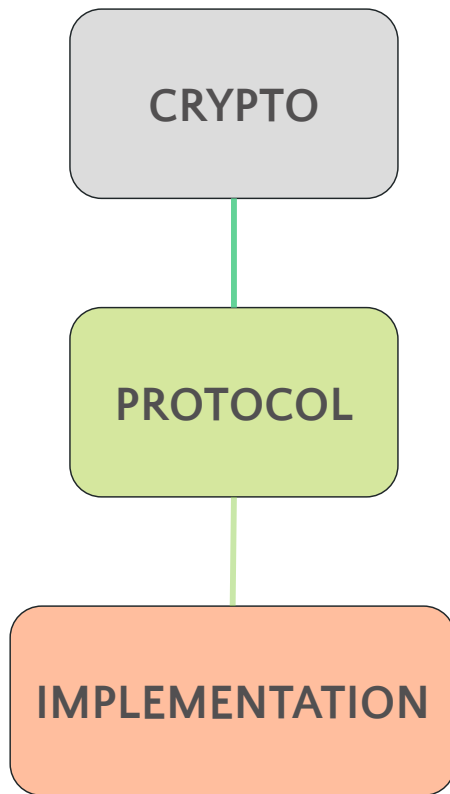
- Build in agility where possible
- Hybrid post-quantum mechanisms provide defense-in-depth

Formally prove protocols before deployment

- Analyze low-level details like message formats
- Pay attention to multi-version protocols

Implementation bugs can bypass all security

- A need to comprehensively analyze code for correctness, memory safety, side channels, ...
- Pay attention to interfaces between new and old code

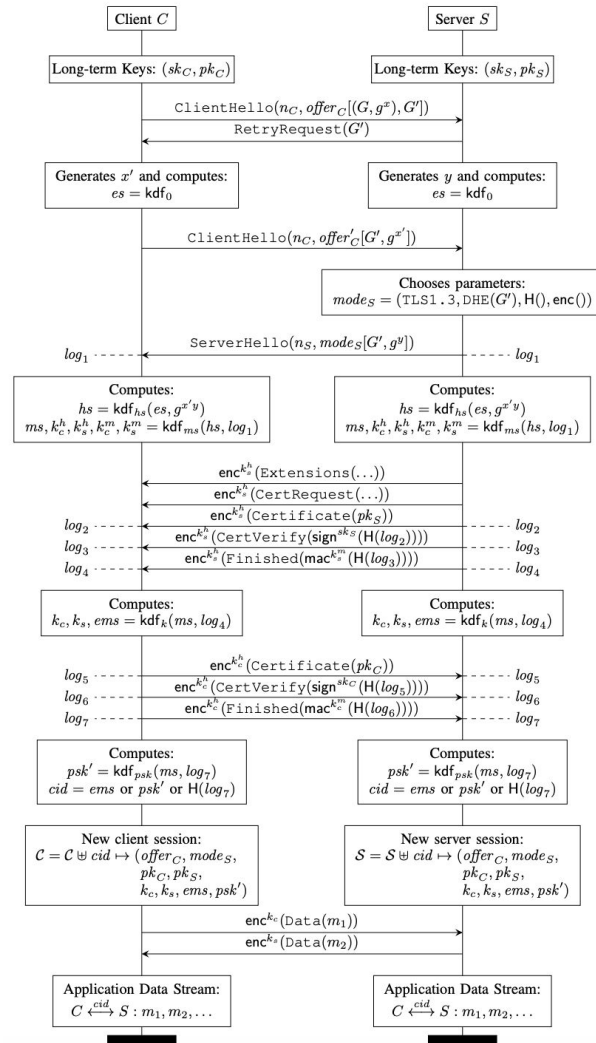


TLS 1.3: Path to standardization

2014

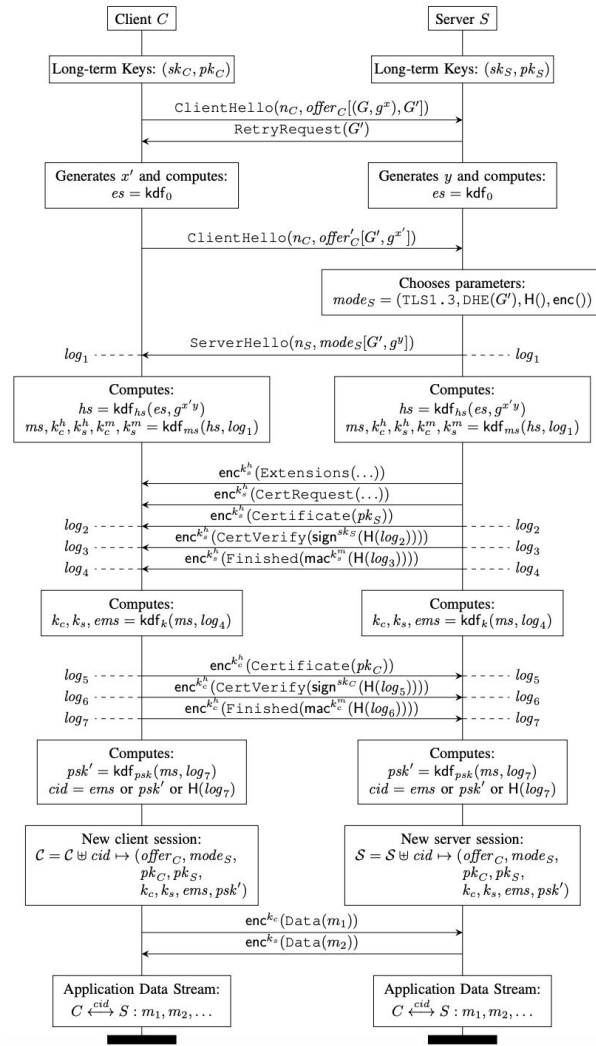
- Draft 5 [Dowling, Fischlin, Günther, Stebila, 2015]
- Draft 7 [Jager, Schwenk, Somorowsky, 2015]
- Draft 9 [Krawczyk, Wee, 2016]
- Draft 10 [Li, Xu, Zhang, Feng, Hu, 2016]
[Fischlin, Günther, Schmidt, Warinschi, 2016]
[Cremers, Horvat, Scott, van der Merwe, 2016]
- Draft 12 [Bhargavan, Brzuska, Fournet, Green, Kohlweiss, Beguelin, 2016]
- Draft 14 [Fischlin, Günther, 2017]
- Draft 18 [Bhargavan, Delignat-Lavaud, Fournet, Kohlweiss, Pan, Protzenko, Rastogi, Swamy, Zanella-Béguelin, Zinzindohoué, 2017]
[Bhargavan, Blanchet, Kobeissi, 2017]
- Draft 21 [Cremers, Horvat, Hoyland, Scott, van der Merwe, 2017]

2018



TLS 1.3: Lessons and Impact

- Strong collaboration between WG and researchers
 - Many pen-and-paper proofs
 - Some **machine-checked proofs** using Tamarin, ProVerif, CryptoVerif, F*
 - Some **formally verified implementations** in C, F*, Rust
 - Proofs now often required for new protocols
- IETF Working Groups
 - **LAKE**: Key exchange protocol for IoT
 - **TLS**: Encrypted Client Hello, KEM-TLS
 - **MLS**: Secure group messaging
- Industrial Protocols
 - PQ3 (iMessage), PQXDH (Signal)



Analyzing Post-Quantum Protocols: Signal

The (Classical) Signal Protocol

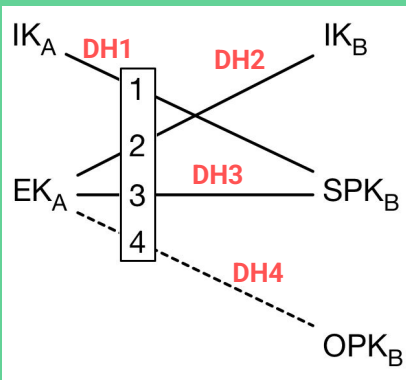
Two parts:

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

Important security guarantees:

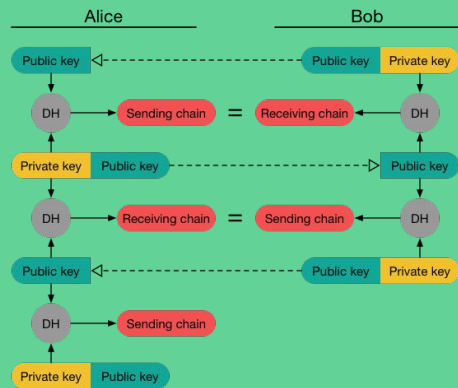
- Mutual authentication
- Post-compromise security
- Forward secrecy
- Deniability

X3DH

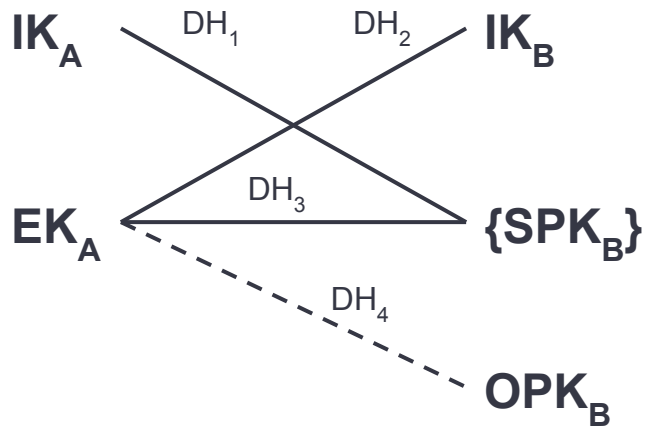


$$SK = KDF(DH1 \parallel DH2 \parallel DH3 \parallel DH4)$$

Double Ratchet

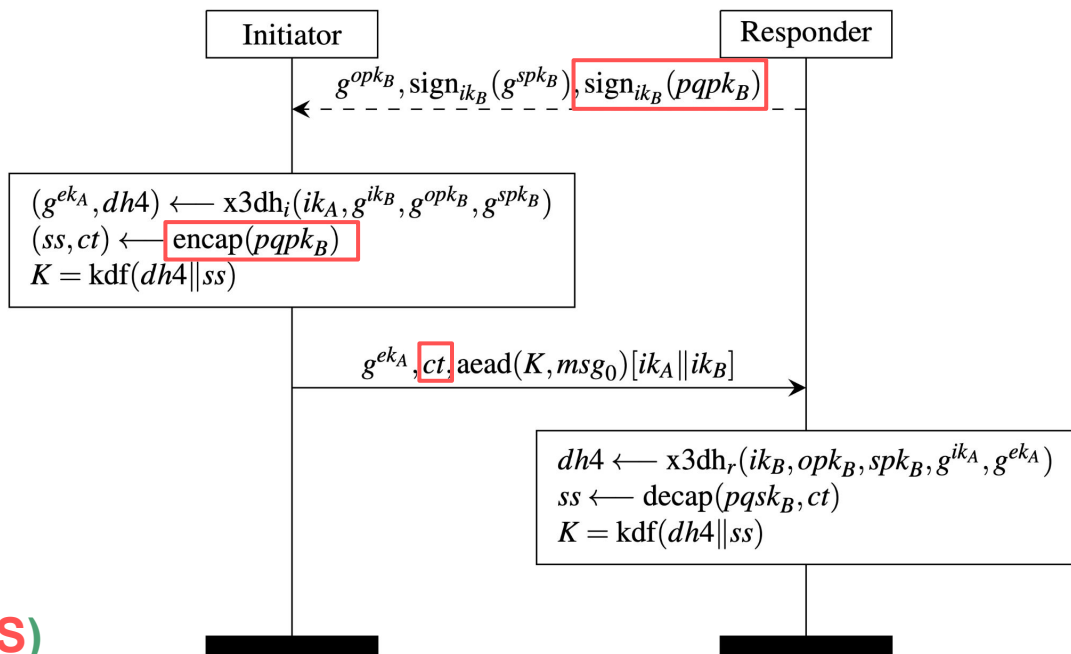


PQXDH Design: Add a PQ-KEM to X3DH



$(SS, CT_{KEM}) \leftarrow \{PQPK_B\}$

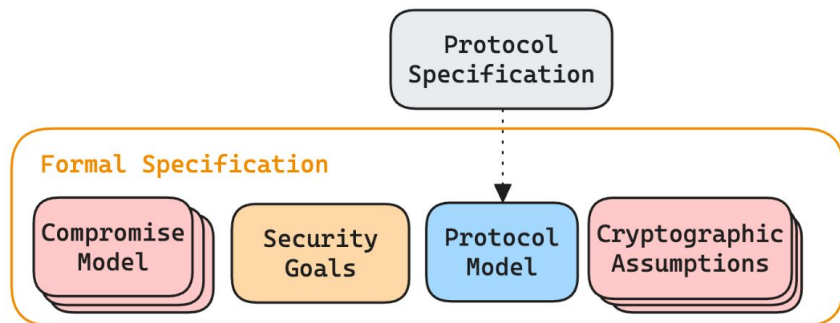
$SK = KDF(DH_1 \parallel DH_2 \parallel DH_3 \parallel DH_4 \parallel SS)$



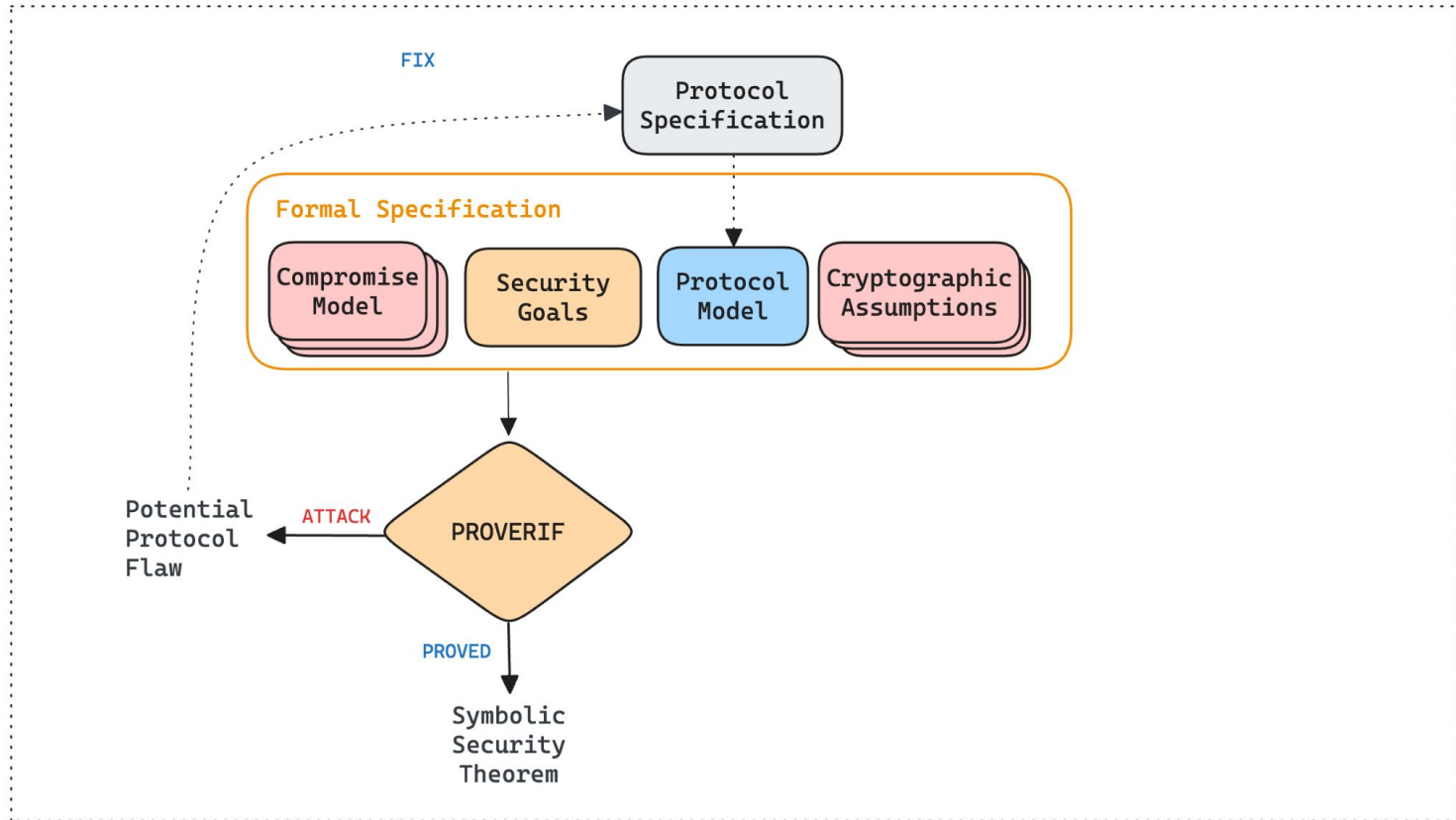
Analyzing PQXDH

- PQXDH is a very small addition to X3DH.
- X3DH has been comprehensively analyzed in a variety of security models
 - Mutual Authentication, Confidentiality, (a form of) Forward Secrecy
- Is PQXDH as secure as X3DH?
- Is it secure against a Harvest-Now-Decrypt-Later quantum adversary?

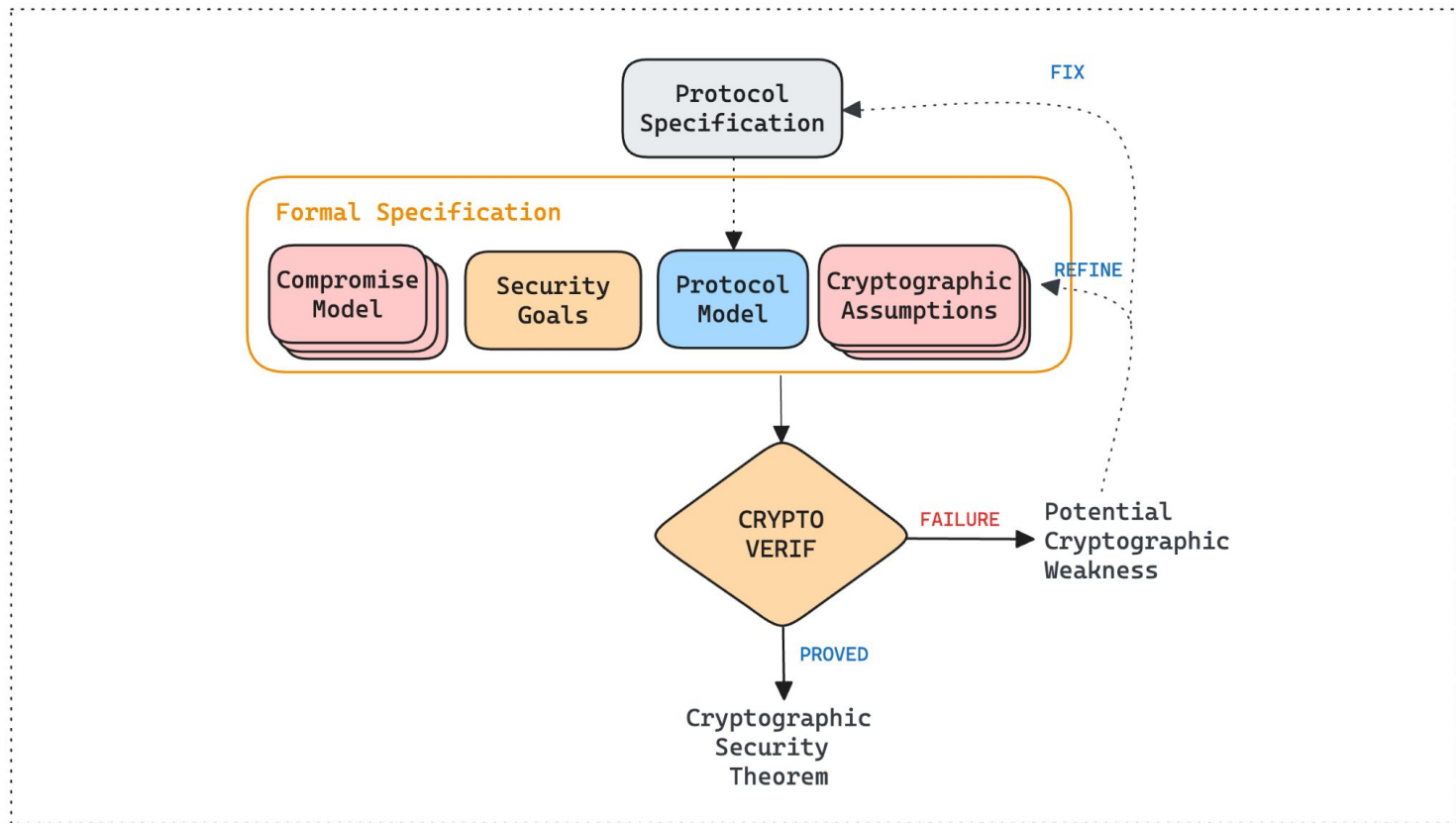
Our Formal Verification Methodology



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 as trace properties about protocol events & attacker knowledge
- Indistinguishability, privacy stated as equivalence properties between processes

Fully automated analysis

- Finds attacks and produces traces
- If no attack found in model, establishes a 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 `table(identity_pubkeys(b,SMUL(IK_s_2,G)))`.

...

Game-Based Proofs with CryptoVerif

Computational crypto model

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

Process, security query syntax like ProVerif

- Secrecy, authentication, indistinguishability

Proofs as a 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

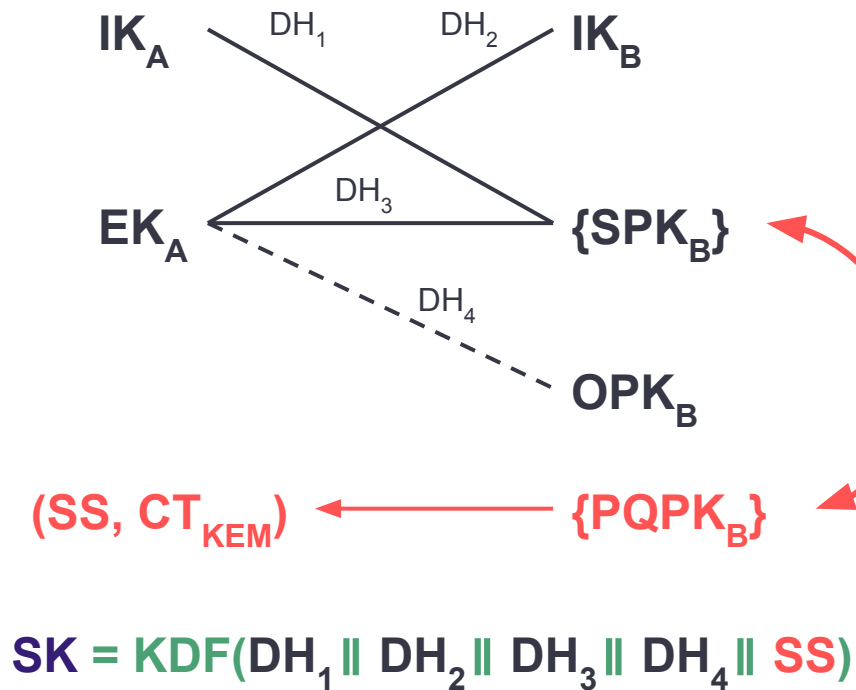
Passive Quantum Adversary Model (Harvest-Now-Decrypt-Later)

- We allow adversary to break certain crypto primitives (e.g. 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



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 classical modes and post-quantum crypto modes in the same protocol.

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

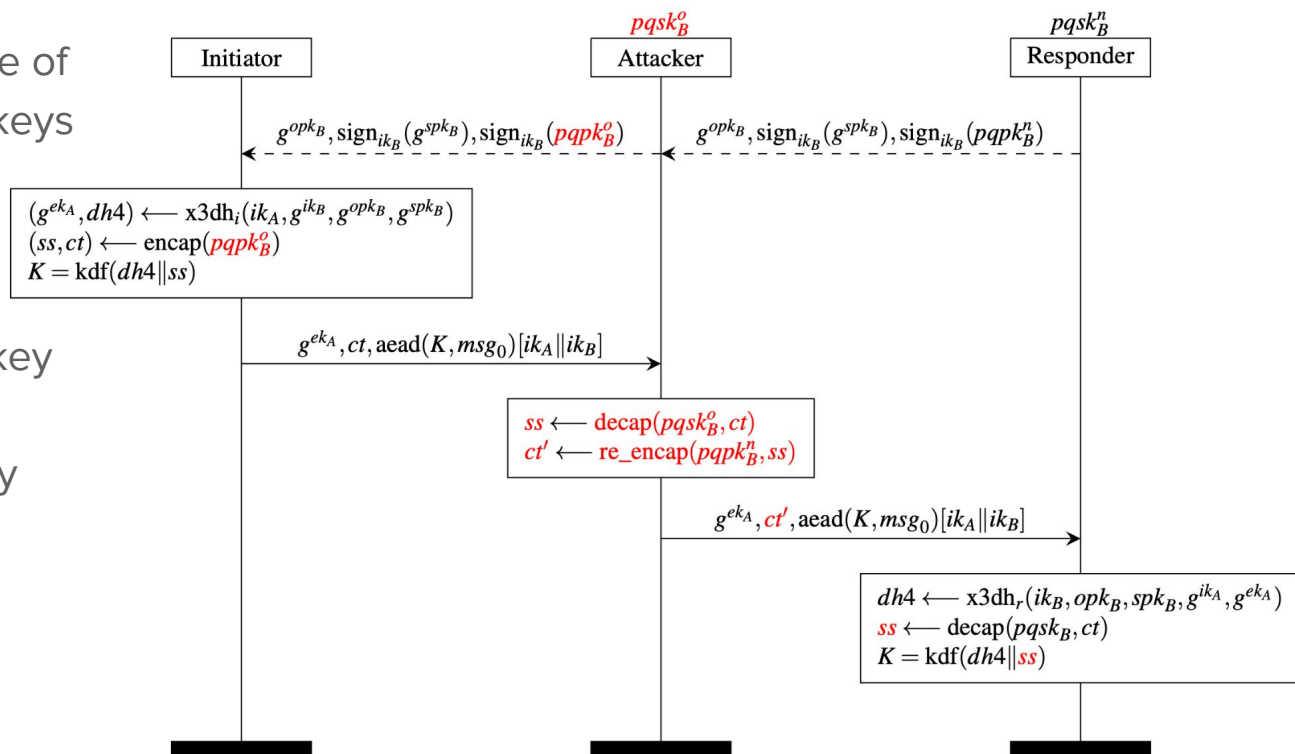
Signal implementation already does this

KEM Re-encapsulation Vulnerability in PQXDH

1. Attacker compromises one of responder's old PQ-KEM keys
2. Attacker provides old PQ-KEM key to initiator
3. Initiator encapsulates SS to (compromised) old key
4. Attacker re-encapsulates SS to responder's new key
5. Responder thinks initiator used new PQ public key

Breaks agreement query

(non-matching transcripts)



PQXDH Revision and Security Theorems

The findings and discussions with Signal team led to a new revision of the 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.

This whole process: spec, analysis, fix, proof, new spec took 1 calendar month.

Formal Analysis for the Full PQ Signal Protocol

New PQ ratcheting protocol (SPQR)

- Uses chunked messages
- Requires a new streaming API for ML-KEM

Formally verified protocol design

- Symbolic analysis with ProVerif
- Dozen variants analyzed

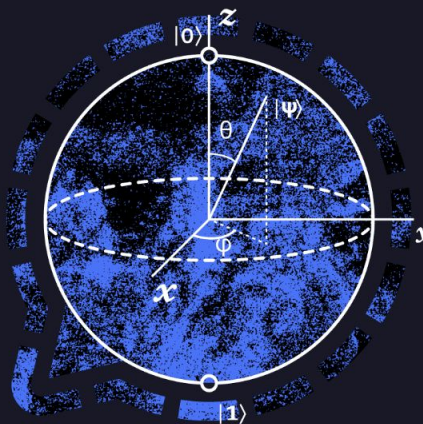
Formal verification for protocol code

- Libcrux implementation of ML-KEM
- Proofs of panic-freedom and correctness using hax under CI



Signal Protocol and Post-Quantum Ratchets

Graeme Connell and Rolfe Schmidt on 02 Oct 2025



Lessons for PQ Automotive Protocols

Standard crypto often needs new APIs

- Constraints on latency, message size
- Formal proofs can help optimize

Analyze protocols as early as possible

- Dozen variants quickly considered
- Bugs found and addressed early

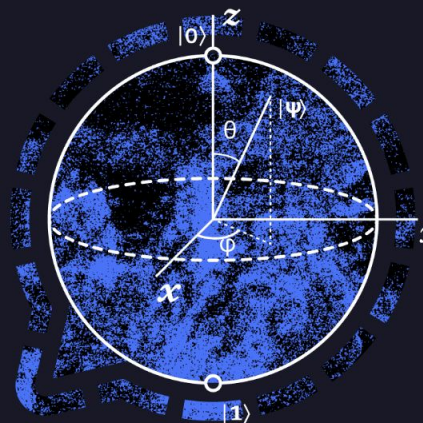
Software verification built into code

- Maintaining invariants becomes natural
- Developers routinely find and fix bugs



Signal Protocol and Post-Quantum Ratchets

Graeme Connell and Rolfe Schmidt on 02 Oct 2025



Towards a High-Assurance Post-Quantum Cryptographic Stack for Automotive Systems

Formally verified cryptographic libraries

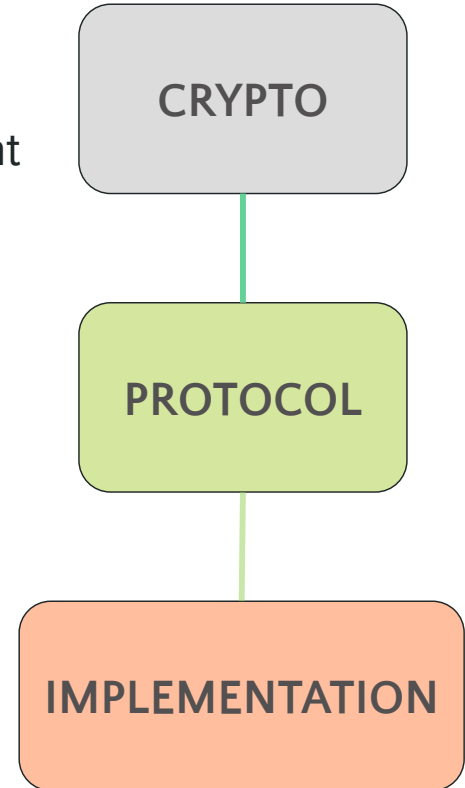
- Memory safe, functionally correct, side-channel resistant
- Easy to Certify, easy to integrate in existing systems

Provably secure post-quantum protocols

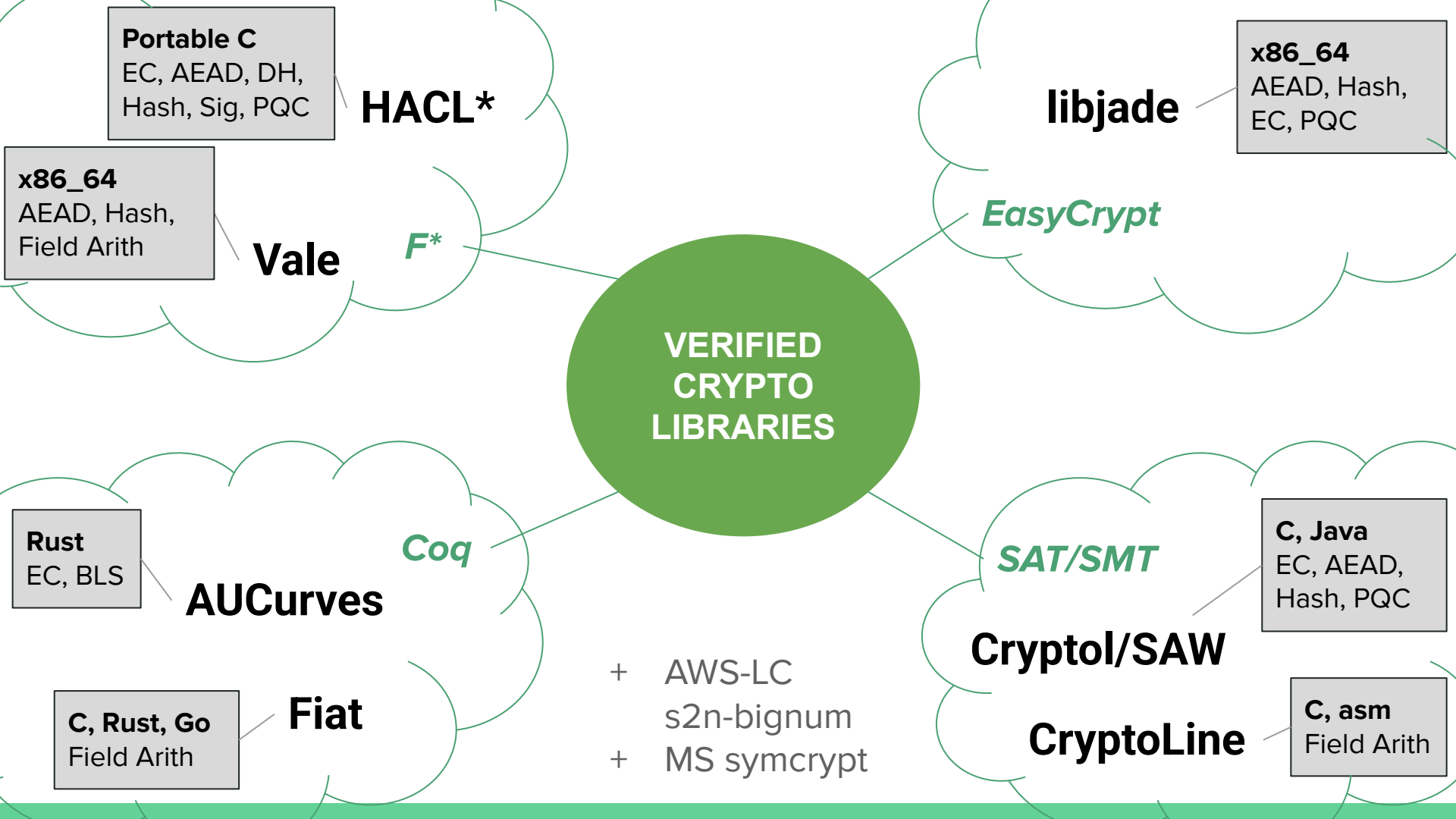
- Secure against classical and quantum adversaries
- Safe and smooth post-quantum transition

High-assurance protocol implementations

- Conforms to protocol spec, complies with regulations
- Strong guarantees for full software supply chain



Verifying Cryptographic Libraries



**VERIFIED
CRYPTO
LIBRARIES**

HACL*

Portable C
EC, AEAD, DH,
Hash, Sig, PQC

libjade

x86_64
AEAD, Hash,
EC, PQC

EasyCrypt

Vale

*F**

x86_64
AEAD, Hash,
Field Arith

SAT/SMT

Cryptol/SAW

CryptoLine

C, Java
EC, AEAD,
Hash, PQC

C, asm
Field Arith

Coq

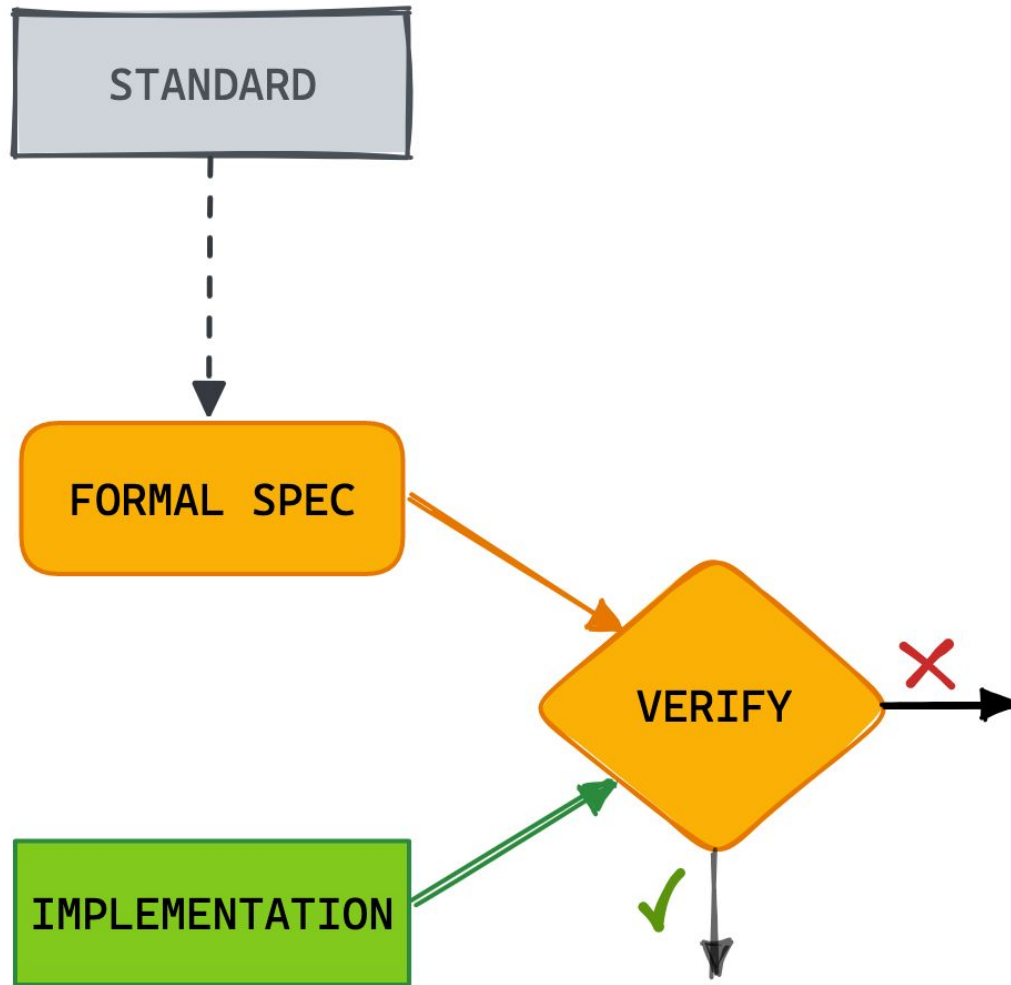
AUCurves

Rust
EC, BLS

Fiat

C, Rust, Go
Field Arith

- + AWS-LC
- + s2n-bignum
- + MS symcrypt



Verified Cryptography Workflow

STANDARD

FORMAL SPEC

IMPLEMENTATION

Internet Research Task Force (IRTF)
Request for Comments: 8439
Obsoletes: [7539](#)
Category: Informational
ISSN: 2070-1721

Y. Nir
Dell EMC
A. Langley
Google, Inc.
June 2018

ChaCha20 and Poly1305 for IETF Protocols

Abstract

This document defines the ChaCha20 stream cipher and the Poly1305 authenticator, both as standard and as a "combined mode", or Authenticated Encryption.

**IETF RFC or
NIST Standard**

2.1. The ChaCha Quarter Round

The basic operation of the ChaCha algorithm is the quarter round. It operates on four 32-bit unsigned integers, denoted a , b , c , and d . The operation is as follows (in C-like notation):

```
a += b; d ^= a; d <<= 16;
c += d; b ^= c; b <<= 12;
a += b; d ^= a; d <<= 8;
c += d; b ^= c; b <<= 7;
```

**In English +
Pseudocode**

2.1.1. Test Vector for the ChaCha Quarter Round

For a test vector, we will use the same numbers as in the example, adding something random for c .

```
a = 0x11111111
b = 0x01020304
c = 0x9b8d6f43
d = 0x01234567
```

+ Test Vectors

STANDARD



FORMAL SPEC

IMPLEMENTATION

```
let line (a:idx) (b:idx) (d:idx) (s:rotval U32) (m:state) : Tot state =  
  let m = m.[a] ← (m.[a] +. m.[b]) in  
  let m = m.[d] ← ((m.[d] ^.  
m.[a]) <<<. s) in m
```

```
let quarter_round a b c d : Tot shuffle =  
  line a b d (size 16) @  
  line c d b (size 12) @  
  line a b d (size 8) @  
  line c d b (size 7)
```

F* Spec
(HACL*)

```
proc chacha20_line(a : int, b : int, d : int, s : int, st : State) = {  
  var state;  
  state <- st;  
  state.[a] <- ((state).[a]) + ((state).[b]);  
  state.[d] <- ((state).[d]) ^ ((state).[a]);  
  state.[d] <- rotate_left ((state).[d]) (s);  
  return state;  
}
```

```
proc chacha20_quarter_round(a : int, b : int, c : int, d : int, st : State) = {  
  var state;  
  state <@ chacha20_line (a, b, d, 16, st);  
  state <@ chacha20_line (c, d, b, 12, state);  
  state <@ chacha20_line (a, b, d, 8, state);  
  state <@ chacha20_line (c, d, b, 7, state);  
  return state;  
}
```

EasyCrypt Spec
(libjade)

STANDARD



FORMAL SPEC



F or Coq or
EasyCrypt...*

VERIFY



Potential Implementation Bug

- Memory Safety Violation
- Functional Correctness Flaw
- Side Channel Vulnerability



Fix and re-verify

IMPLEMENTATION



Deploy Code

Verified Cryptography Workflow

Good news: For any modern crypto algorithm, there is probably a verified implementation

- You don't have to sacrifice **performance**
- **Mechanized proofs** that you can run and re-run yourself
- You (mostly) don't have to read or understand the proofs
- Formally verified crypto in **NSS, BoringSSL, aws-lc, ...**

Verifying Post-Quantum Crypto in Rust

FIPS 203

Federal Information Processing Standards Publication

Module-Lattice-Based Key-Encapsulation Mechanism Standard

Category: Computer Security

Subcategory: Cryptography

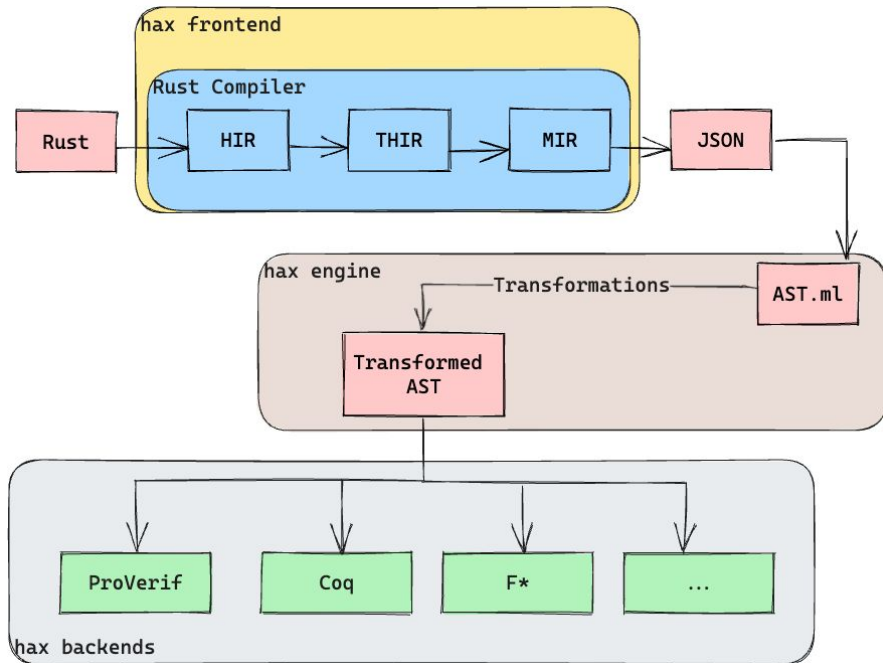
Information Technology Laboratory
National Institute of Standards and Technology
Gaithersburg, MD 20899-8900

This publication is available free of charge from:
<https://doi.org/10.6028/NIST.FIPS.203>

Published August 13, 2024

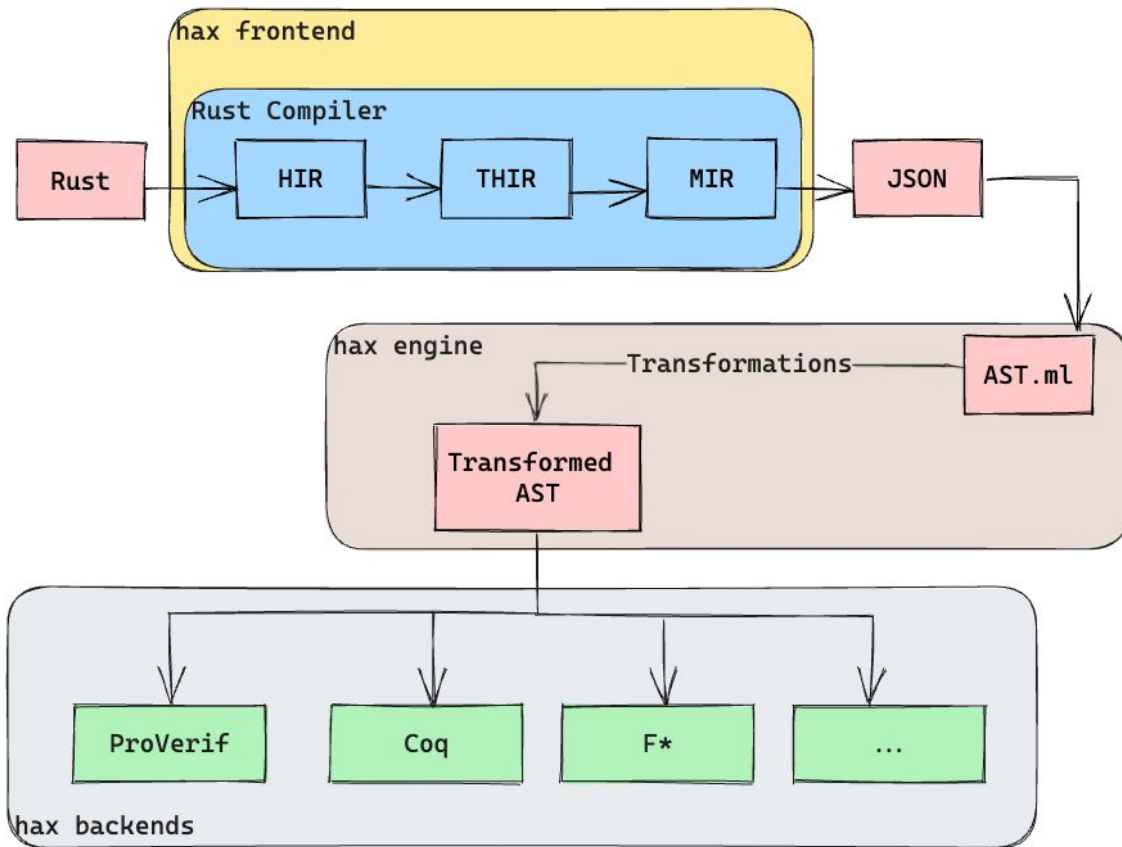
Hax:

- Verify Rust code using multiple provers
- Applied to cryptographic libraries
 - Runtime Safety
 - Functional Correctness
 - Secret Independence
- New: Support more Rust, Lean backend



```
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
}
```

hax: linking Rust code with proof backends



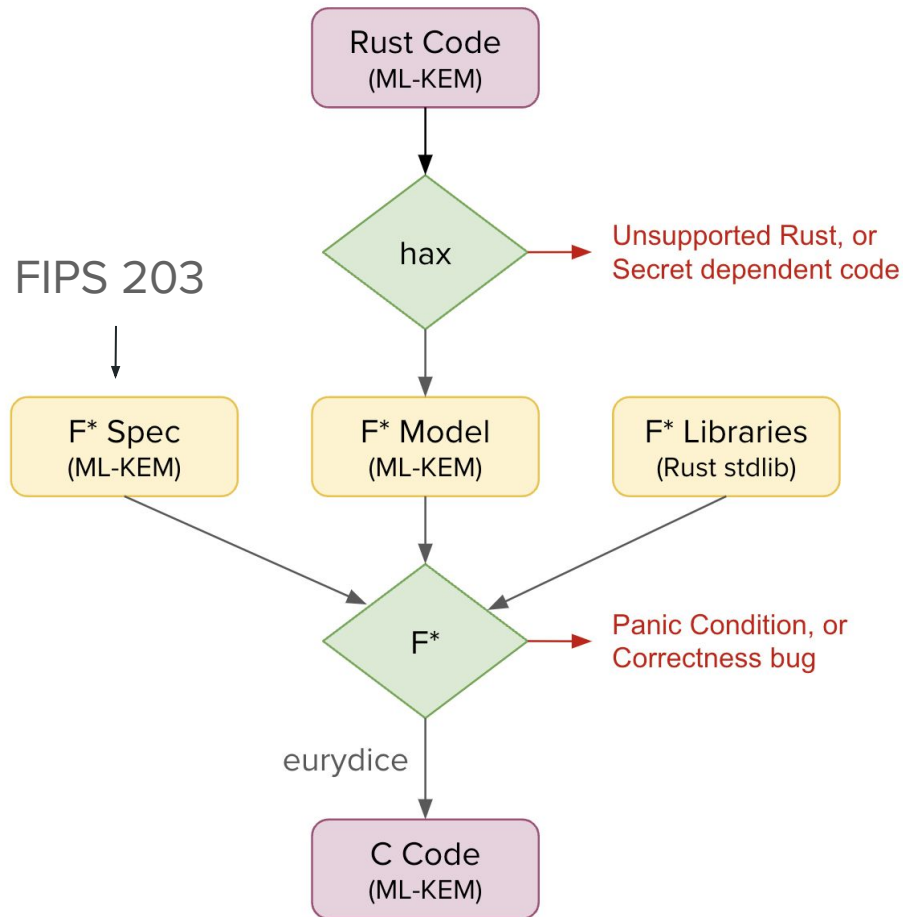
~~CRYSPEX~~



AARHUS
UNIVERSITY

Inria

Verifying crypto code written in Rust and C using `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, 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: $\text{result} \% 3329 \approx \text{input} \% 3329$
 $\&\& -3329 < \text{result} < 3329$

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 timing bugs **at Rust source level**.

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

KyberSlash: a new timing vulnerability

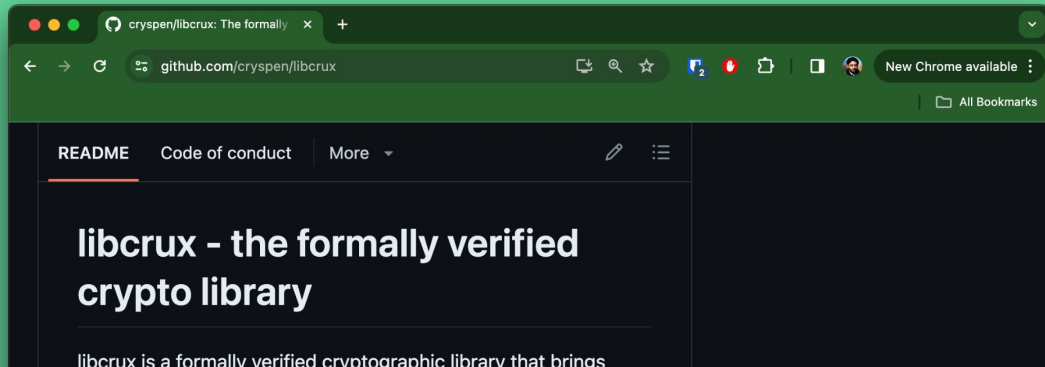
```
void poly_tomsg(uint8_t msg[KYBER_INDCPA_MSGBYTES], const poly *a)
{
    unsigned int i,j;
    uint16_t t;

    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, ...

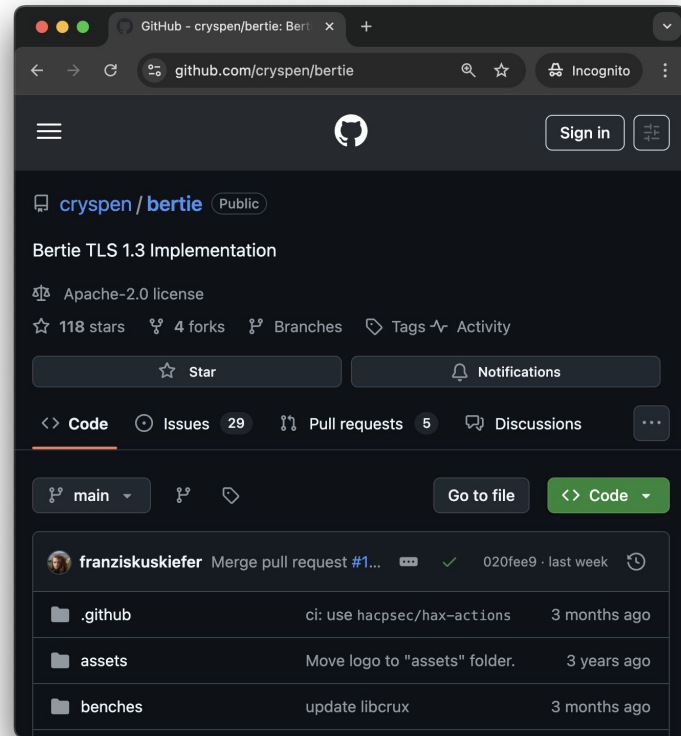
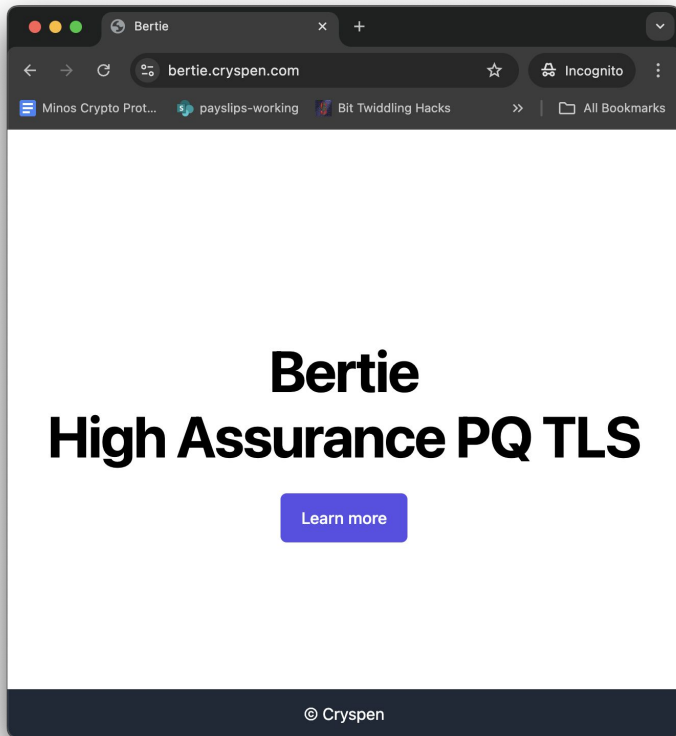
Bug found during
Formal Verification
of our Rust code!

Libcrux has an **optimized, portable**, formally **verified** implementation of ML-KEM and ML-DSA in **Rust and C**. Our ML-KEM code is now deployed in Firefox, OpenSSH, Signal, ...

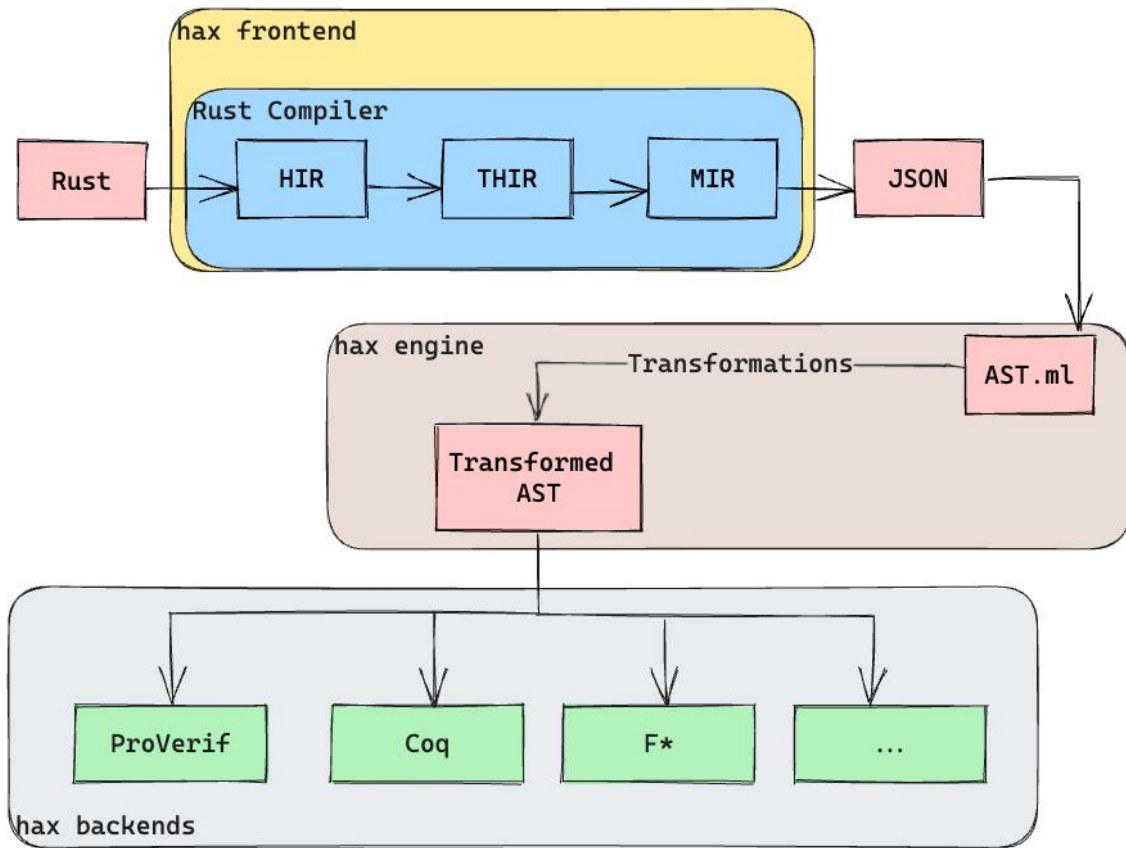


Verifying Post-Quantum Protocol Code

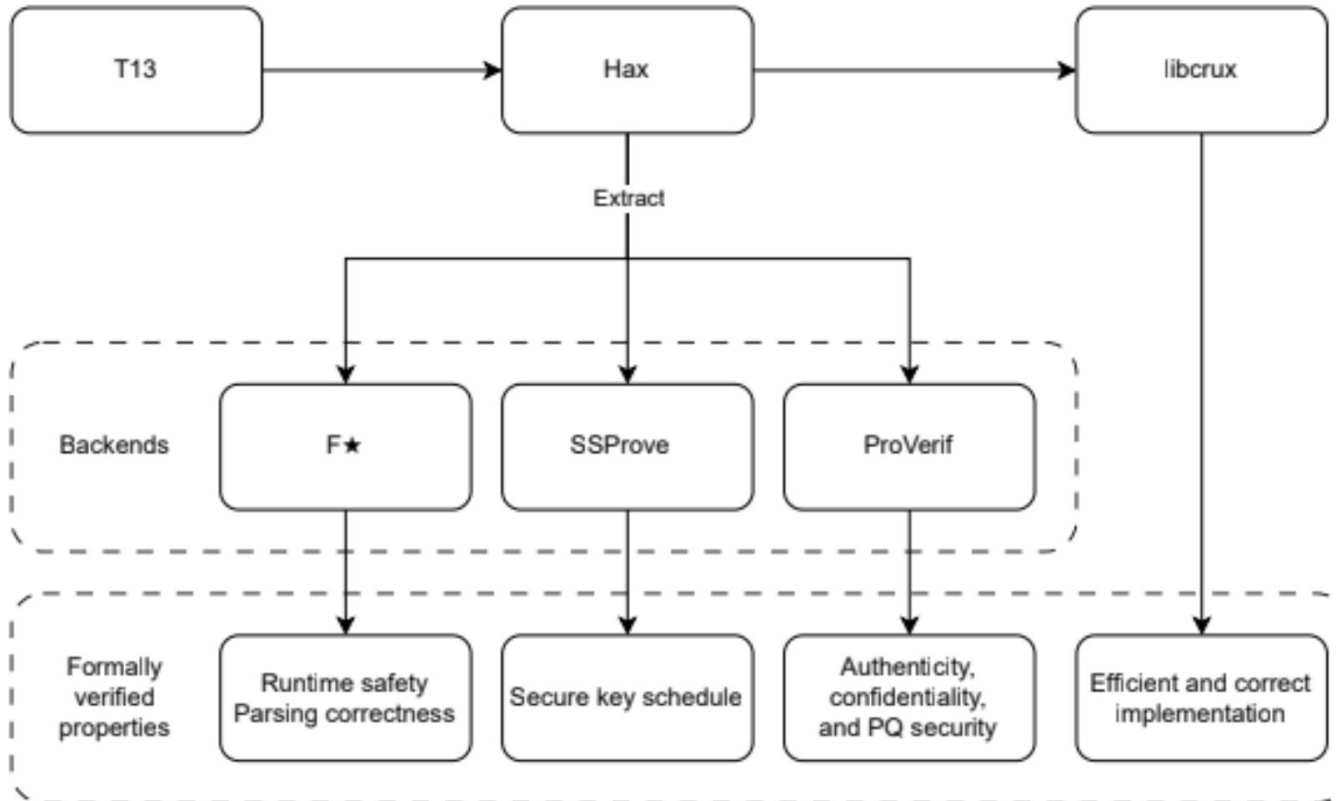
BERT13: Verified PQ-TLS 1.3 in Rust



hax: bridging the gap between code and proof



Verifying BERT13 with hax



Verifying BERT13 using hax

- We use the **Rust type system** to enforce state machine invariants
- We use **libcrux** for verified cryptography
- We use **F*** for properties like parsing correctness and panic-freedom
- We use **ProVerif** to prove protocol security for the TLS 1.3 implementation
- We use **SSProve** to cryptographically verify the key schedule implementation

Table 2: Formal Verification Results for Bert13

Backend Prover	Rust Modules	Rust LoC	Translated LoC	Properties Proven	Time Taken for Proofs (s)
SSProve	1	425	815	Core Key Schedule Security	11m17s
ProVerif	3	1723	5980	Forward Secrecy, Authentication HNDL Post-Quantum Security	20s
F★	8	3264	10964	Runtime Safety, Unambiguous Formats	1m21s

Conclusion

Transitioning to post-quantum protocols requires new formal analysis

- More than just switching the crypto algorithm
- For constrained environments, we often need new crypto APIs
- Beware of downgrade and cross-protocol attacks

Fast, verified post-quantum crypto is available and already deployed

- See NSS, BoringSSL, mbedTLS, AWS-LC, MS Symcrypt, ...
- Verifying legacy C and asm is hard, Rust offers a sweet spot

Verifying protocol code can find and prevent large classes of bugs

- Many formal tools, some that protocol developers can use themselves
- Rust-based verification frameworks are becoming practical and effective

Questions?

- Papers: <https://cryspen.com/research/>
- Code:
 - Hax: <https://github.com/cryspen/hax>
 - Libcrux: <https://github.com/cryspen/libcrux>
 - Bertie: <https://github.com/cryspen/bertie>