

Software Verification meets Real-World Cryptography

Karthikeyan Bhargavan

+ many, many collaborators

~~CRYSPEN~~

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

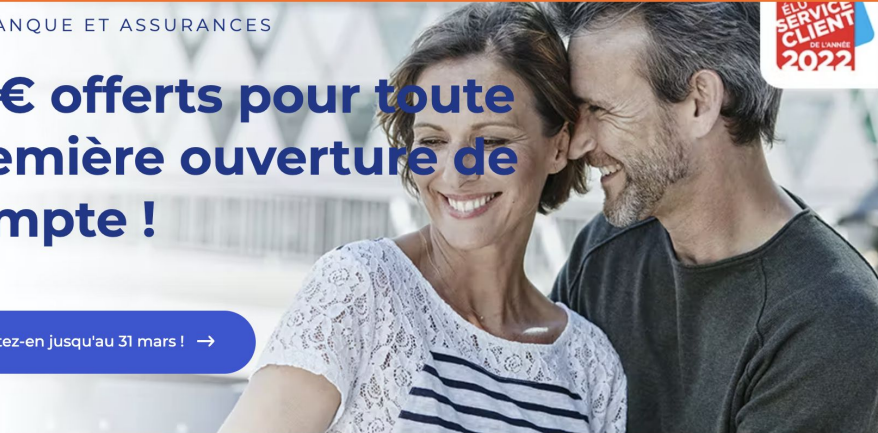
Découvrir LCL

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

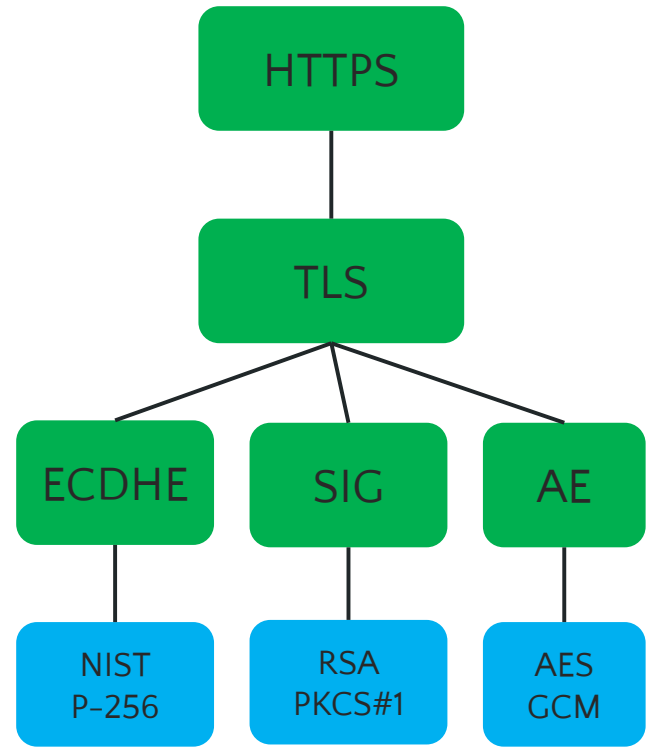
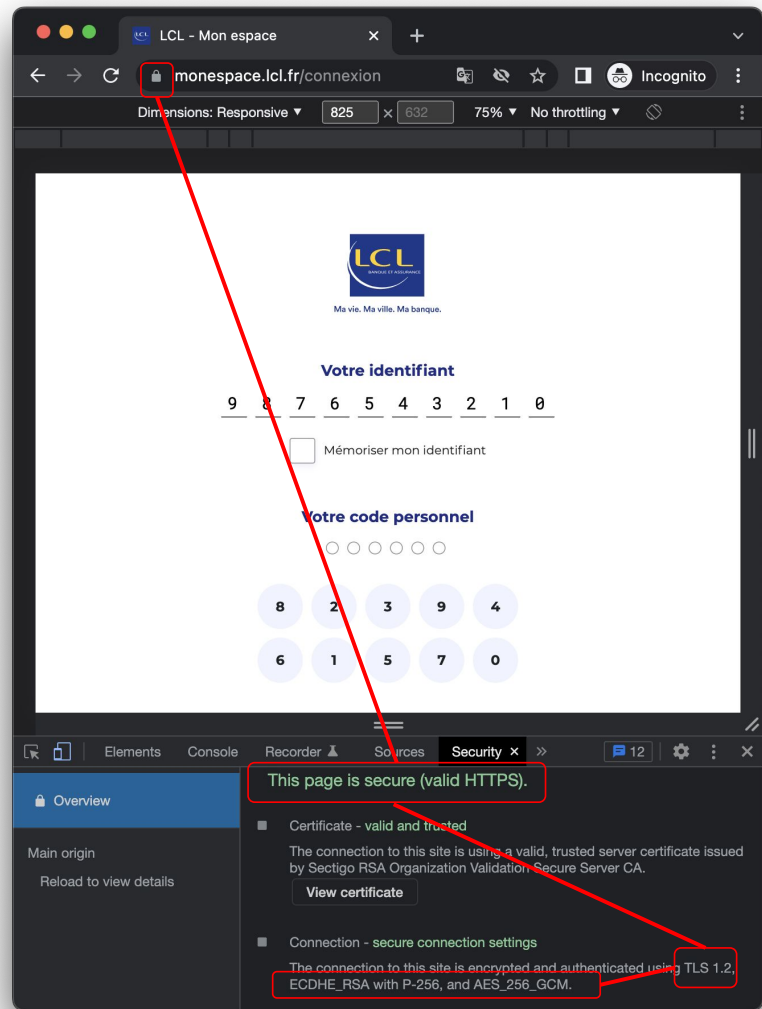
LCL BANQUE ET ASSURANCES

40€ offerts pour toute première ouverture de compte !

Profitez-en jusqu'au 31 mars ! →



Parlons de vos besoins



- TLS is well-studied since 1994!
- So, is this a secure connection?

Many Practical Attacks on TLS (until TLS 1.3)

- **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 often goes wrong in protocols like TLS?

Crypto Weaknesses

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

Protocol Design Flaws

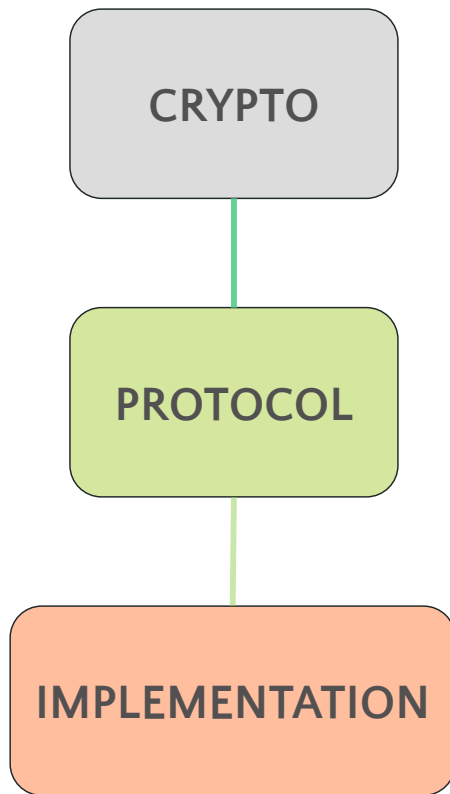
- Downgrade attacks, Transcript collisions

Implementation Bugs

- Parsing, State machines, Memory safety

Often, a mix of all the above!

A need for formal verification tools that can scale up to large cryptographic protocol implementations.



Formal Methods for Crypto

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

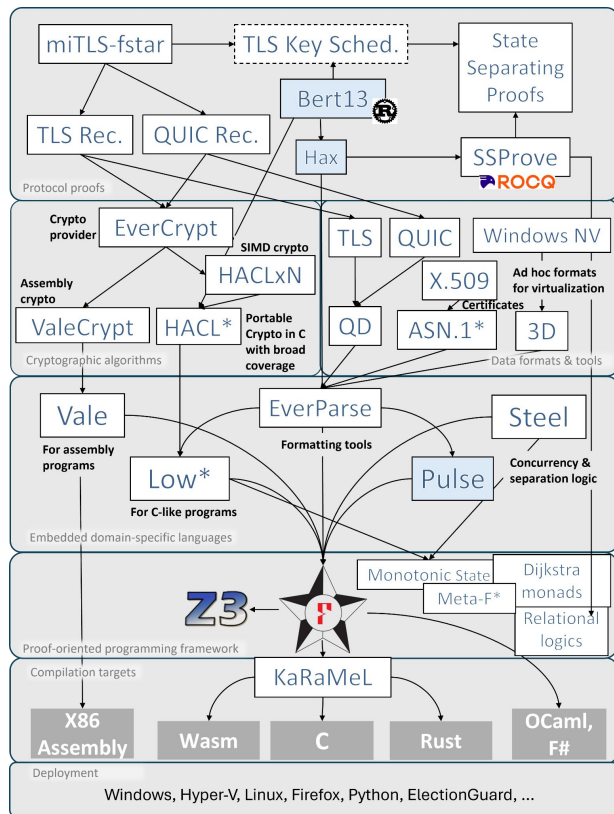
“Applying formal, **machine-checkable** approaches to the design, analysis, and implementation of cryptography.”

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

Project Everest [2016-21]



~~CRYS~~PEN [2021-]

~~CRYS~~PEN
☰

High Assurance Software

Gain confidence in your most critical systems with Cryspen's advanced tools and technologies for provable security and reliability.

Verifying Cryptographic Libraries

OpenSSL
Cryptography and SSL/TLS Toolkit

NSS

BoringSSL

Web

**CRYPTO
LIBRARIES**

Lang



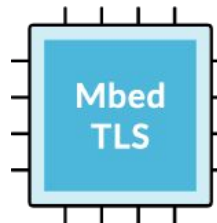
OS



IoT



wolfSSL



OpenSSL
Cryptography and SSL/TLS Toolkit

6

NSS

BoringSSL

Web

**TRUSTED
COMPUTING
BASE**

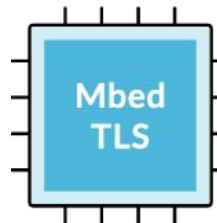
Lang



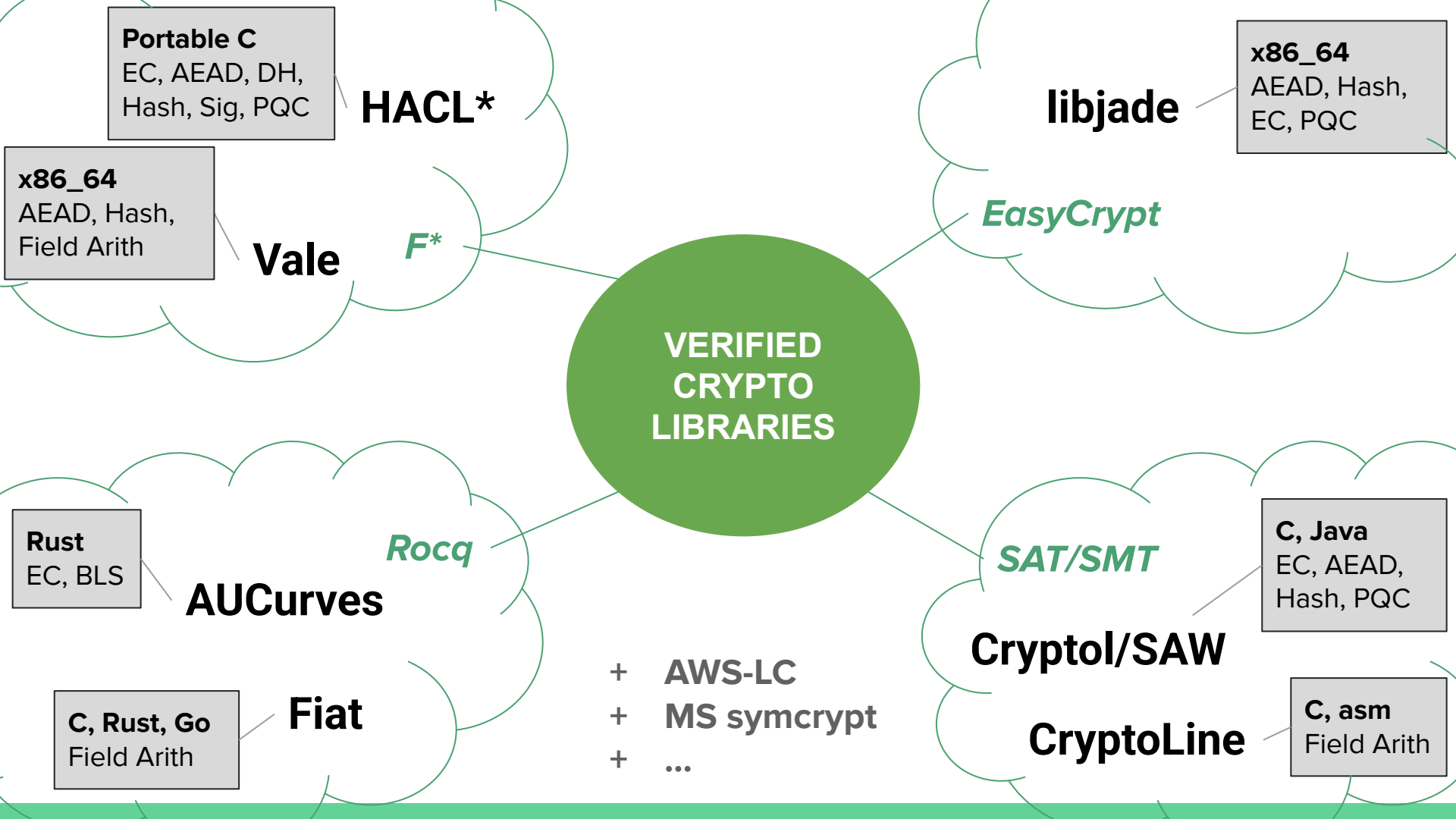
OS



IoT

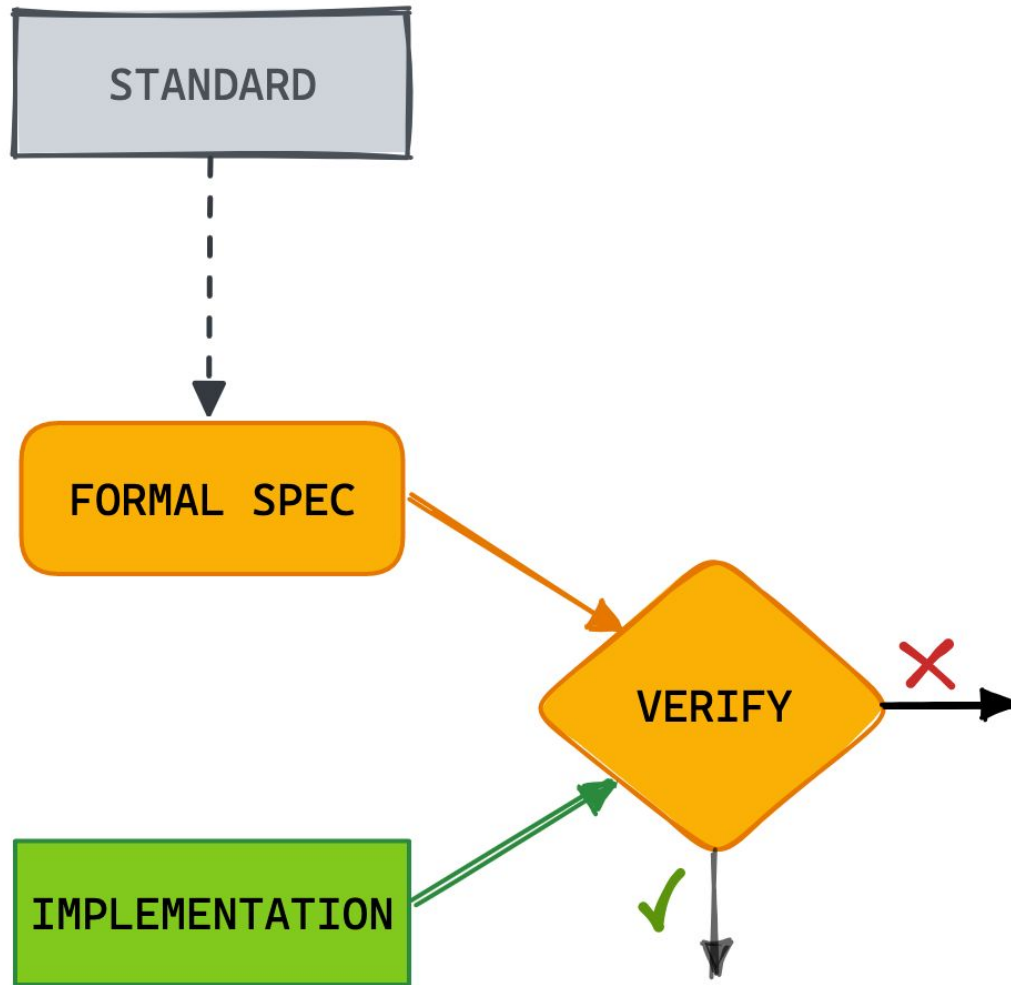


12



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
- Formally verified crypto in **Firefox, Chrome, Linux, AWS, ...**



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
```

Succinct,
Mathematical
Functions

```
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

IMPLEMENTATION

```
let line st a b d r =  
  let sta = st.(a) in  
  let stb = st.(b) in  
  let std = st.(d) in  
  let sta = sta +. stb in  
  let std = std ^. sta in  
  let std = rotate_left std r in  
  st.(a) ← sta;  
  st.(d) ← std
```

```
let quarter_round st a b c d =  
  line st a b d (size 16);  
  line st c d b (size 12);  
  line st a b d (size 8);  
  line st
```

F* (Low*)
Implementation

Stateful, Optimized,
“Constant-time”,
Portable, C-like code

Translate

```
static inline void quarter_round(uint32_t *st, uint32_t a, uint32_t b, uint32_t c, uint32_t d)  
{  
  uint32_t sta = st[a];  
  uint32_t stb0 = st[b];  
  uint32_t std0 = st[d];  
  uint32_t sta10 = sta + stb0;  
  uint32_t std10 = std0 ^ sta10;  
  uint32_t std2 = std10 << (uint32_t)16U | std10 >> (uint32_t)16U;  
  st[a] = sta10;  
  st[d] = std2;  
  ...  
}
```

Portable C Code

STANDARD



FORMAL SPEC

IMPLEMENTATION

```
inline fn __line_ref(reg u32[16] k,  
                    inline int a b c r)  
    -> reg u32[16]  
{  
    k[a] += k[b];  
    k[c] ^= k[a];  
    _, _, k[c] = #ROL_32(k[c], r);  
    return k;  
}  
  
inline fn __quarter_round_ref(reg u32[16] k,  
                             inline int a b c d)  
    -> reg u32[16]  
{  
    k = __line_ref(k, a, b, d, 16);  
    k = __line_ref(k, c, d, b, 12);  
    k = __line_ref(k, a, b, d, 8);  
    k = __line_ref(k, c, d, b, 7);  
    return k;  
}
```

Jasmin
Implementation

Translate

Intel AVX2
Assembly

Stateful, Optimized,
“Constant-time”,
assembly-like code

```
vpaddq    %ymm4, %ymm0, %ymm0  
vpxor     %ymm0, %ymm12, %ymm12  
vpshufb   (%rsp), %ymm12, %ymm12  
vpaddq    %ymm12, %ymm8, %ymm8  
vpaddq    %ymm6, %ymm2, %ymm2  
vpxor     %ymm8, %ymm4, %ymm4  
vpxor     %ymm2, %ymm14, %ymm14  
vpslld    $12, %ymm4, %ymm15  
vpsrld    $20, %ymm4, %ymm4  
vpxor     %ymm15, %ymm4, %ymm4  
vpshufb   (%rsp), %ymm14, %ymm14  
vpaddq    %ymm4, %ymm0, %ymm0  
vpaddq    %ymm14, %ymm10, %ymm10  
vpxor     %ymm0, %ymm12, %ymm12  
vpxor     %ymm10, %ymm6, %ymm6  
vpshufb   32(%rsp), %ymm12, %ymm12  
vpslld    $12, %ymm6, %ymm15  
vpsrld    $20, %ymm6, %ymm6  
...
```

STANDARD



FORMAL SPEC



F or Rocq or
EasyCrypt...*

VERIFY



Potential Implementation Bug

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



Fix and re-verify

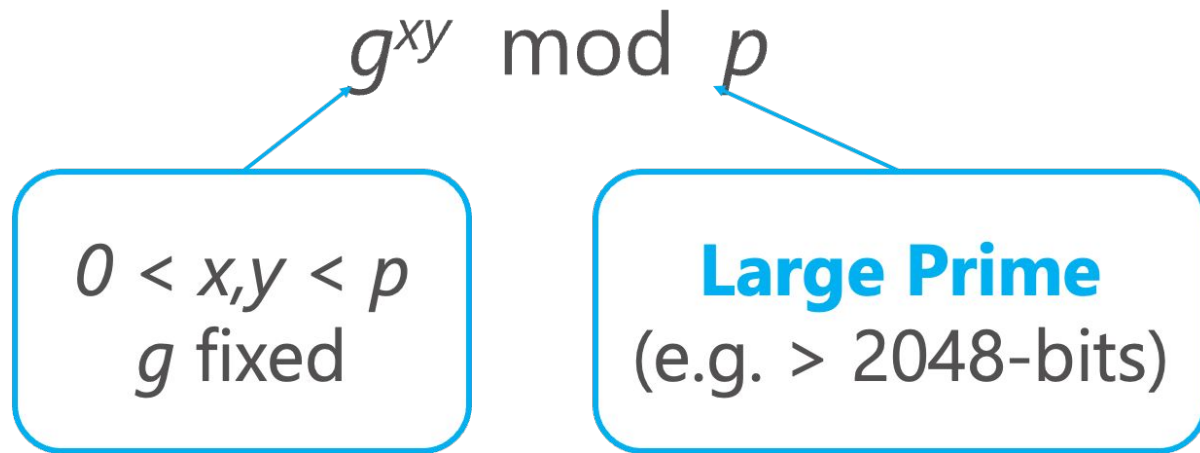
IMPLEMENTATION



Deploy Code

Verified Cryptography Workflow

Example: Diffie-Hellman Computation



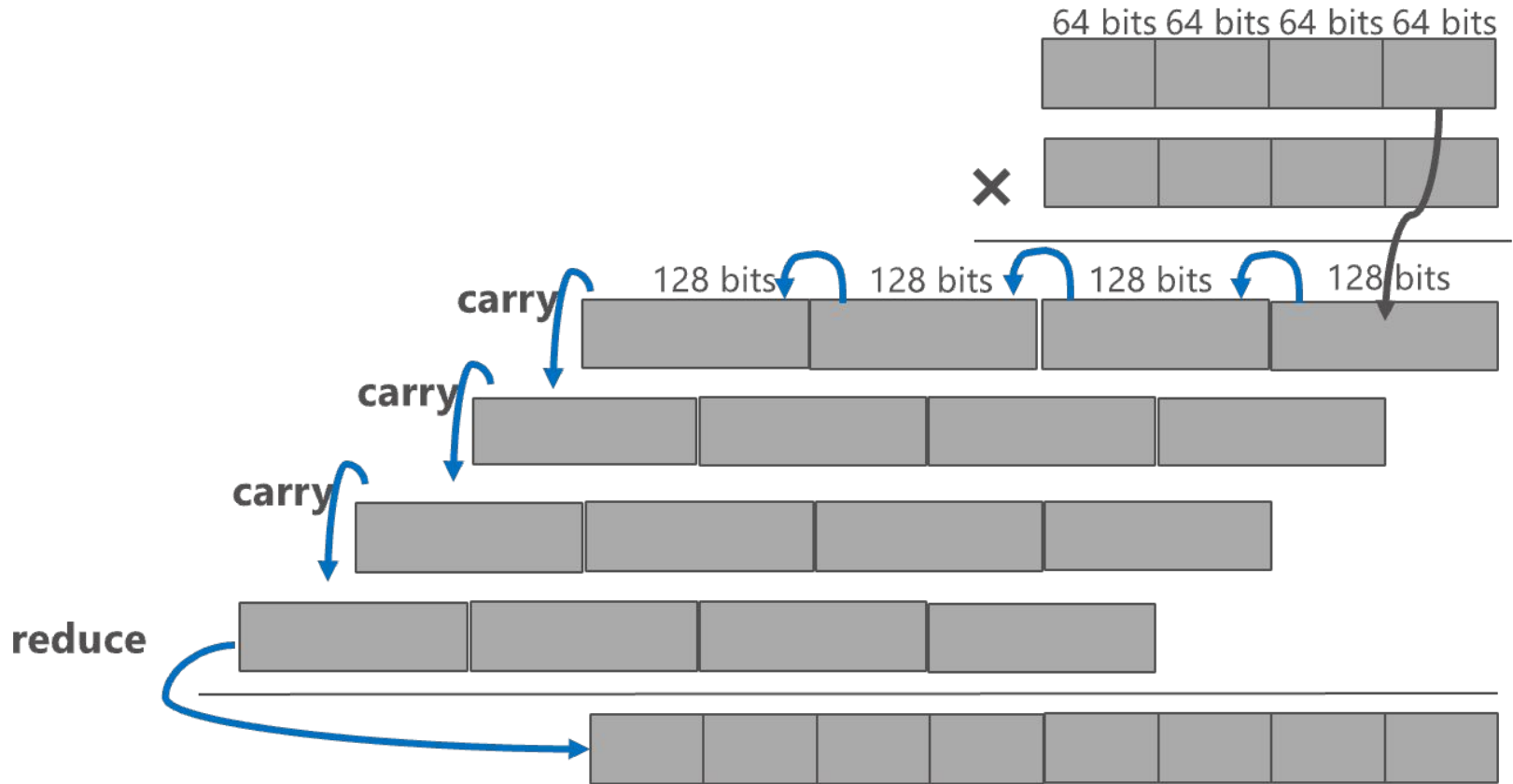
Prime field arithmetic, implemented using modular multiplication
Can often be the **performance bottleneck** on a Web server/IoT device

Textbook Multiplication (a * b)

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

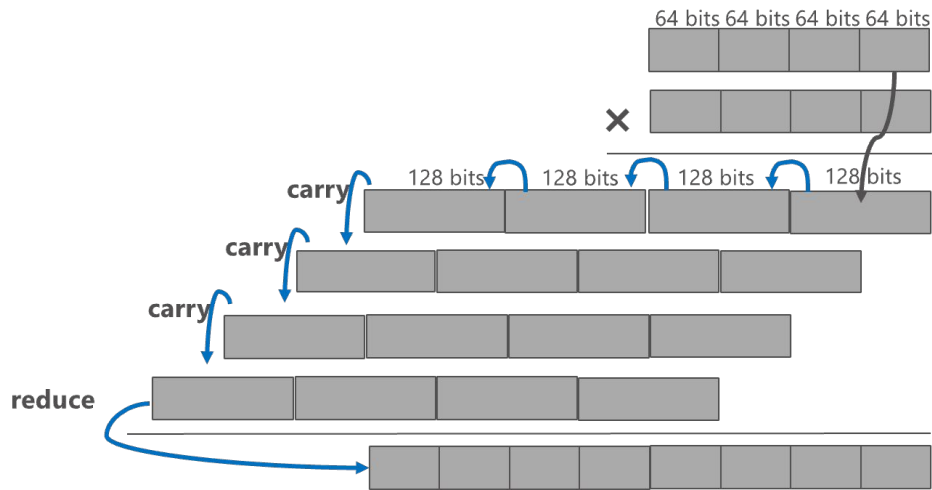
$$\begin{array}{r} 1101 \\ \times 1010 \\ \hline 0000 \\ 1101 \\ 0000 \\ + 1101 \\ \hline 10000010 \end{array}$$

256-bit modular multiplication ($a * b \% p$)



Common Bugs in Modular Arithmetic

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



Low-probability bugs, hard to find by testing.

Side-channel attacks

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

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

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!

Verifying Modular Arithmetic in F^*

```
val fmul:
  out:lbuffer uint64 4
→ f1:lbuffer uint64 4
→ f2:lbuffer uint64 4
→ tmp:lbuffer uint64 16
→ Stack unit
  (requires λ h →
    live h out ∧ live h f1 ∧ live h f2 ∧ live h tmp ∧
    mutually_disjoint [out; f1; f2; tmp])
  (ensures λ h0 _ h1 →
    modifies (loc out | + | loc tmp) h0 h1 ∧
    feval h1 out == Spec.fmul (feval h0 f1) (feval h0 f2))
```

Secret Integers
(constant-time ops)



Memory Safety
Pre-conditions

Memory Safety
Post-condition

Correctness
Specification

Low* Specification for 256-bit modular multiplication

- Verified portable C implementation in Low*
- Verified x64 assembly implementation in Vale

HACL*, Vale, and EverCrypt [2017-2024]

Verified crypto library

- Includes most modern crypto algorithms
- Generates C, Rust, wasm
- Deployed in NSS, Linux, WireGuard, Python, ...

Large engineering effort

- 245K lines of F* and Vale → 135K lines of C and asm
- 3+ Phds, many researchers
- Continuous integration and proof maintenance

Algorithm	Portable C code	Arm A64	Intel x64			
		Neon	AVX	AVX2	AVX512	Vale
AEAD						
Chacha20-Poly1305 AES-GCM	✓ [43] (+)	✓ (*)	✓ (*)	✓ (*)	✓ (*)	✓ [20]
Hashes						
SHA-224,256	✓ [43] (+)	✓ (*)	✓ (*)	✓ (*)	✓ (*)	✓ [20]
SHA-384,512	✓ [43] (+)	✓ (*)	✓ (*)	✓ (*)	✓ (*)	
Blake2s, Blake2b	✓ [34] (+)	✓ (*)	✓ (*)	✓ (*)		
SHA3-224,256,384,512	✓ [34]					
HMAC and HKDF						
HMAC (SHA-2,Blake2)	✓ [43]	✓ (*)	✓ (*)	✓ (*)	✓ (*)	
HKDF (SHA-2,Blake2)	✓ [43]	✓ (*)	✓ (*)	✓ (*)	✓ (*)	
ECC						
Curve25519	✓ [43]					✓ [34]
Ed25519	✓ [43]					
P-256	✓ [34]					
High-level APIs						
Box	✓ [43]					
HPKE	✓ (*)	✓ (*)	✓ (*)	✓ (*)	✓ (*)	✓ (*)

Lessons from HACL*

Deploying verified crypto in the large

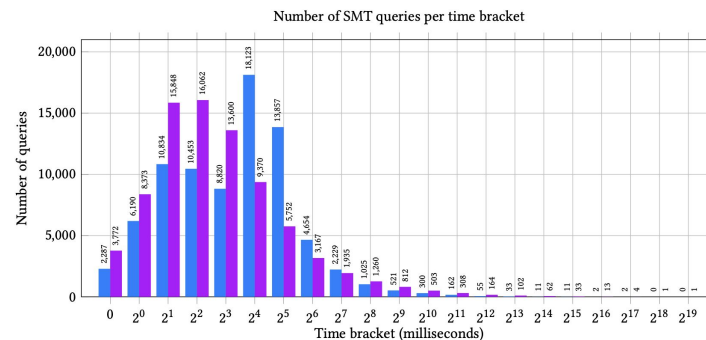
- Performance must match unverified code
- Generated C and assembly must be reviewable
- Need to support flexible APIs, many platforms

Building and maintaining crypto proofs

- Memory safety dominates annotation effort
- SMT automation is crucial for scalability
- A small number of “fragile” proofs can blow up verification time and maintenance effort

Implementation	Radix	Language	CPU cy.
donna64 [2]	51	64-bit C	159634
fiat-crypto [36]	51	64-bit C	145248
amd64-64 [26]	51	Intel x86_64 asm	143302
sandy2x [27]	25.5	Intel AVX asm	135660
EverCrypt portable (<i>this paper</i>)	51	64-bit C	135636
openssl* [5]	64	Intel ADX asm	118604
Oliveira et al. [63]	64	Intel ADX asm	115122
EverCrypt targeted (<i>this paper</i>)	64	64-bit C	113614
		+ Intel ADX asm	

Figure 9. *Performance comparison between Curve25519 Implementations.*



Verifying Post-Quantum Crypto in Rust

CRYSPEN

FIPS 203

Federal Information Processing Standards Publication

Module-Lattice-Based Key-Encapsulation Mechanism Standard

Category: Computer Security

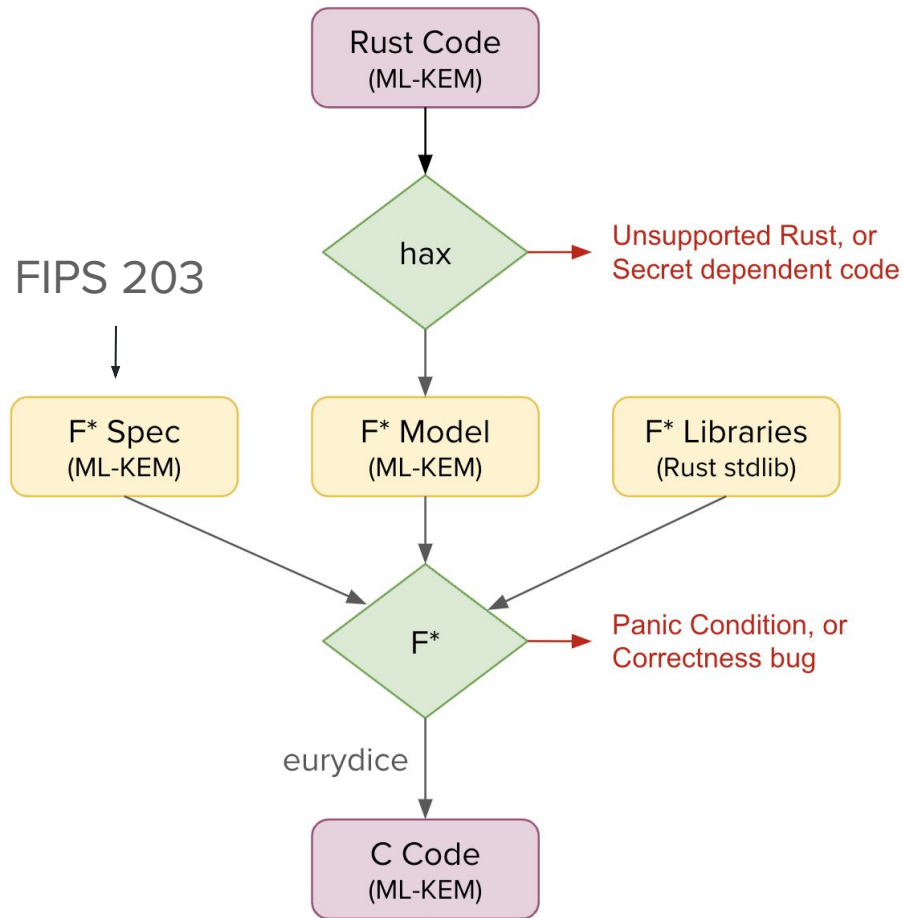
Subcategory: Cryptography

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

Verifying crypto code
written in Rust
using HAX
and 



Writing Crypto Code in Rust

```
fn barrett_reduce(value: i32) -> i32 {  
    let t = i64::from(value) * BARRETT_MULTIPLIER + (BARRETT_R >> 1);  
    let quotient = (t >> BARRETT_SHIFT) as i32;  
    let remainder = value - quotient * FIELD_MODULUS;  
    remainder  
}
```

Barrett Reduction Algorithm

(in constant time, so cannot directly use modulus operator)

Potential Panics in Rust Code

```
fn barrett_reduce(value: i32) -> i32 {  
    let t = i64::from(value) * BARRETT_MULTIPLIER + (BARRETT_R >> 1);  
    let quotient = (t >> BARRETT_SHIFT) as i32;  
    let remainder = value - quotient * FIELD_MODULUS;  
    remainder  
}
```

These arithmetic operations may overflow or underflow causing the code to **crash** or compute **wrong results**

Specifying Panic Freedom and Correctness

```
#[requires(value >= -BARRETT_R && value <= BARRETT_R)]  
#[ensures(|result| result > -FIELD_MODULUS && result < FIELD_MODULUS  
        && result % FIELD_MODULUS == value % FIELD_MODULUS)]  
fn barrett_reduce(value: i32) -> i32 {  
    ...  
    remainder  
}
```

Specify pre-condition and post-condition
(Automatically verified using $F^* + Z3$)

Enforcing Secret Independence

Type-based static analysis to avoid certain side channel attacks

- **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

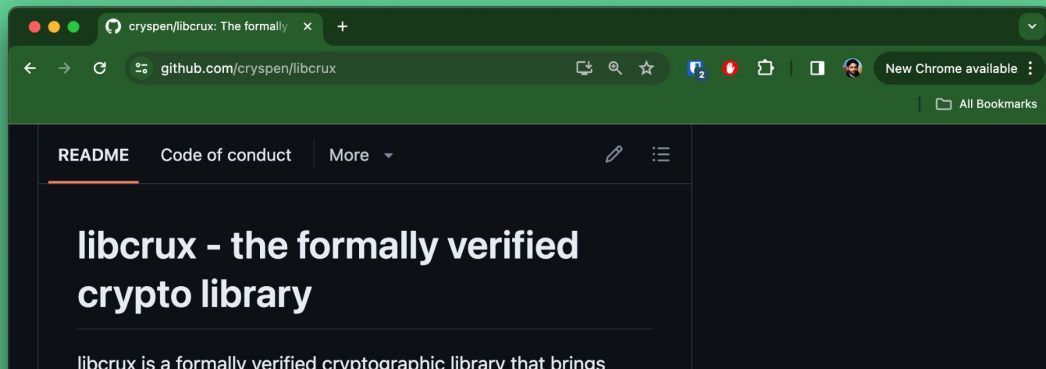
```
void poly_tomsg(uint8_t msg[KYBER_INDCPA_MSGBYTES], const poly *a)
{
    unsigned int i,j;
    uint16_t t;

    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 deployed in
Firefox, OpenSSH, Signal, ...



Lessons and Limitations

Hard to prevent side-channel attacks from Rust

- Platform-specific timing, power analysis, faults, speculative execution
- Compiler may introduce side-channels during optimization
- Needs precise machine models, crypto support from compiler
- **Alternative:** write and verify everything in assembly (Jasmin, Vale)

Verification needs to fit crypto development workflow

- Crypto developers can write Rust or jasmin, but not F* or Rocq
- Standalone C code is easier to integrate into existing C projects
- Testing, fuzzing, on multiple platforms uncovers unexpected bugs!

Verifying Cryptographic Protocol Code

Recall: What often goes wrong in protocols like TLS?

Crypto Weaknesses

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

Protocol Design Flaws

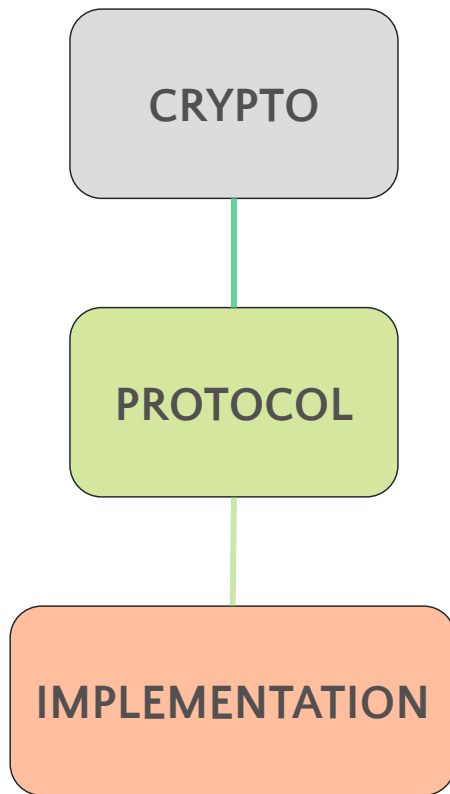
- Downgrade attacks, Transcript collisions

Implementation Bugs

- Parsing, State machines, Memory safety

Often, a mix of all the above!

A need for verification tools that can account for crypto and scale to large protocols like TLS.



The TLS 1.3 Standardization Project

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]

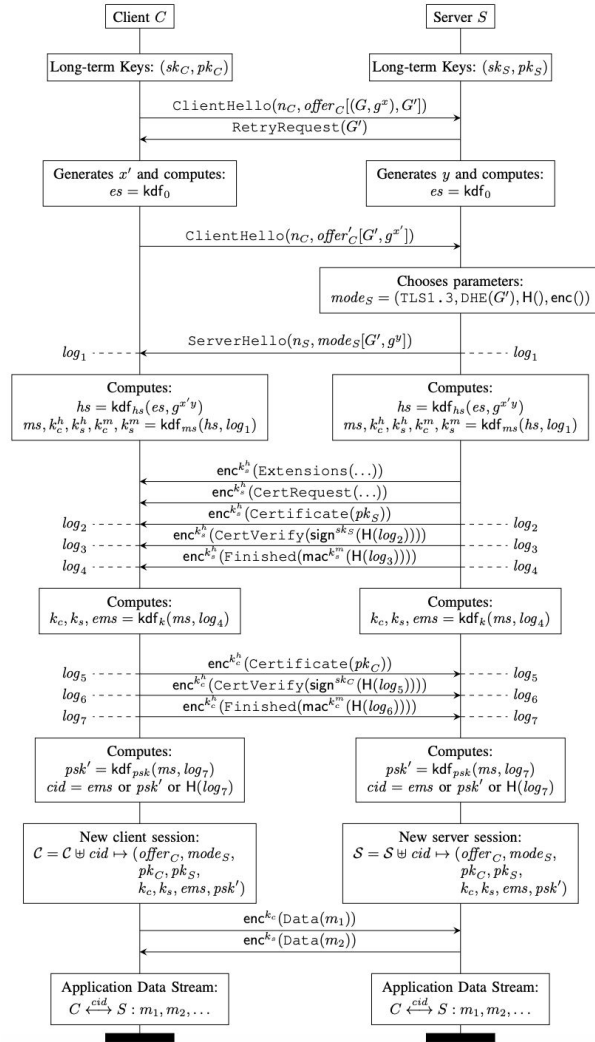
Draft 12 [Bhargavan, Brzuska, Fournet, Green, Kohlweiss, Beguelin, 2016]

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



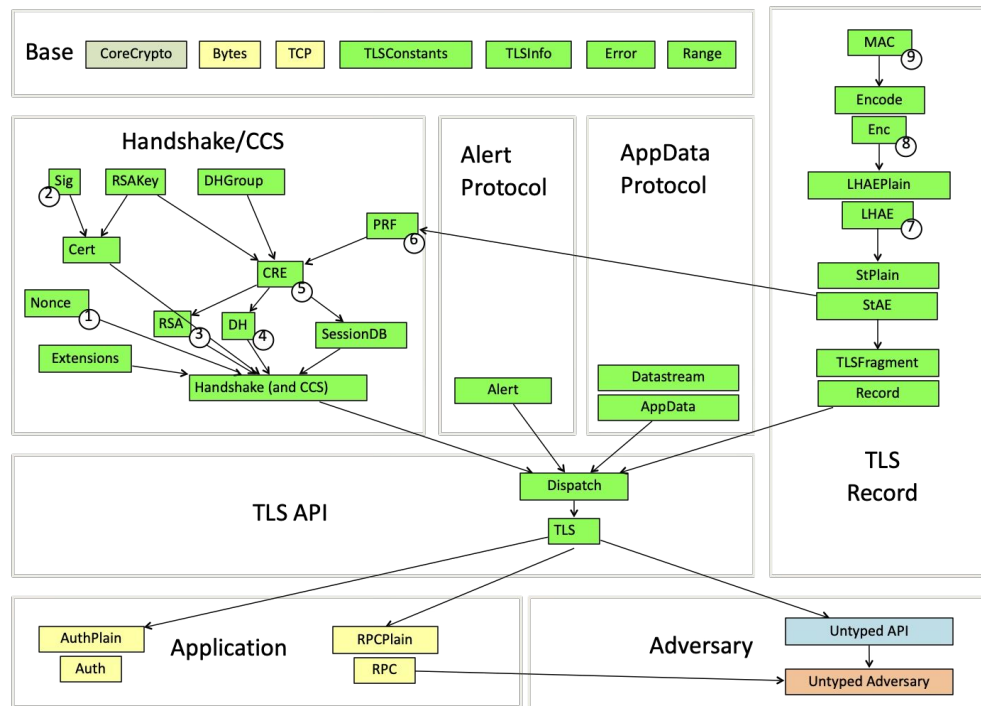
miTLS [2012-2017]

Verified implementation of TLS

- Covers many modes of TLS 1.0-1.3
- State machines, message formats
- Security + Correctness proof in F^*
- Verified crypto from EverCrypt

Major effort, high impact

- 30K+ lines, 100+ modules
- Helped find major attacks on TLS:
Triple Handshake, FREAK, Logjam
- Influenced design of TLS 1.3



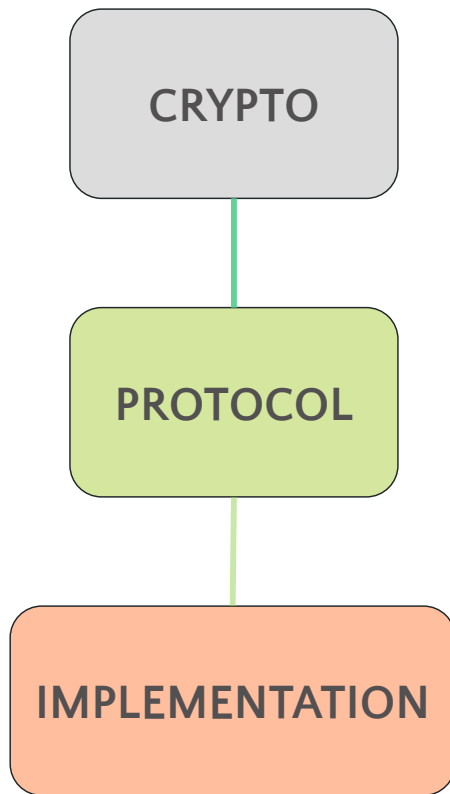
Lessons from miTLS

Large protocol specs, complex distributed systems

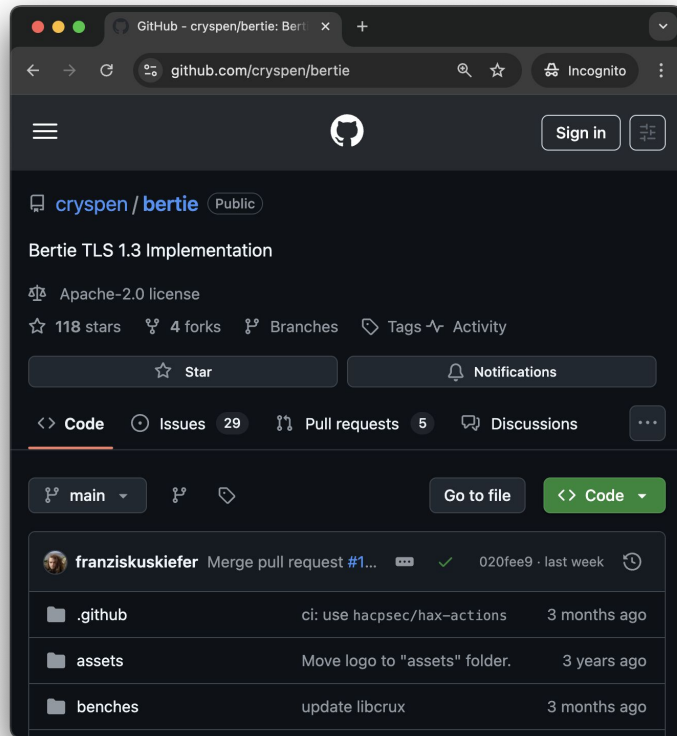
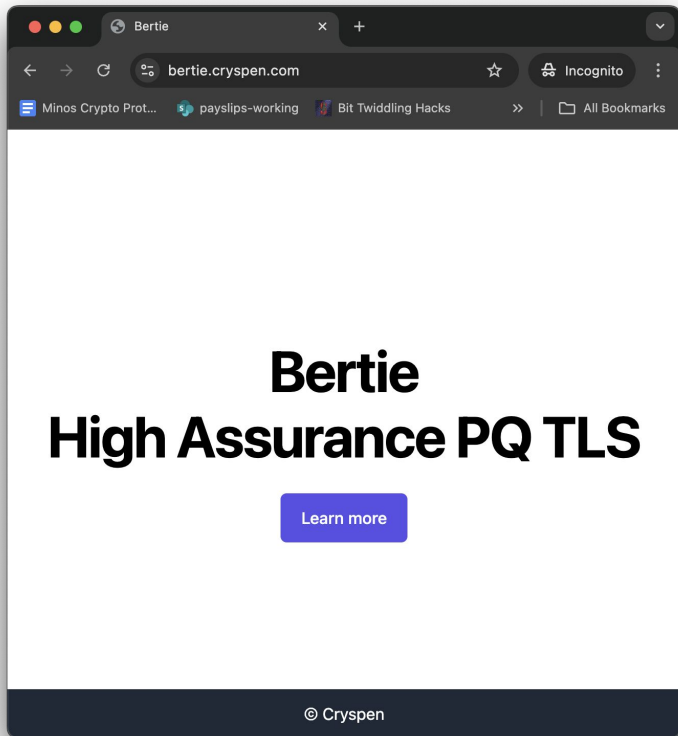
- Not just crypto, **low-level protocol details** matter
- Risks from composition flaws, **downgrade attacks**
- Verifying a reference implementation is a way to ensure we captured all relevant details

Monolithic security+correctness proofs do not scale

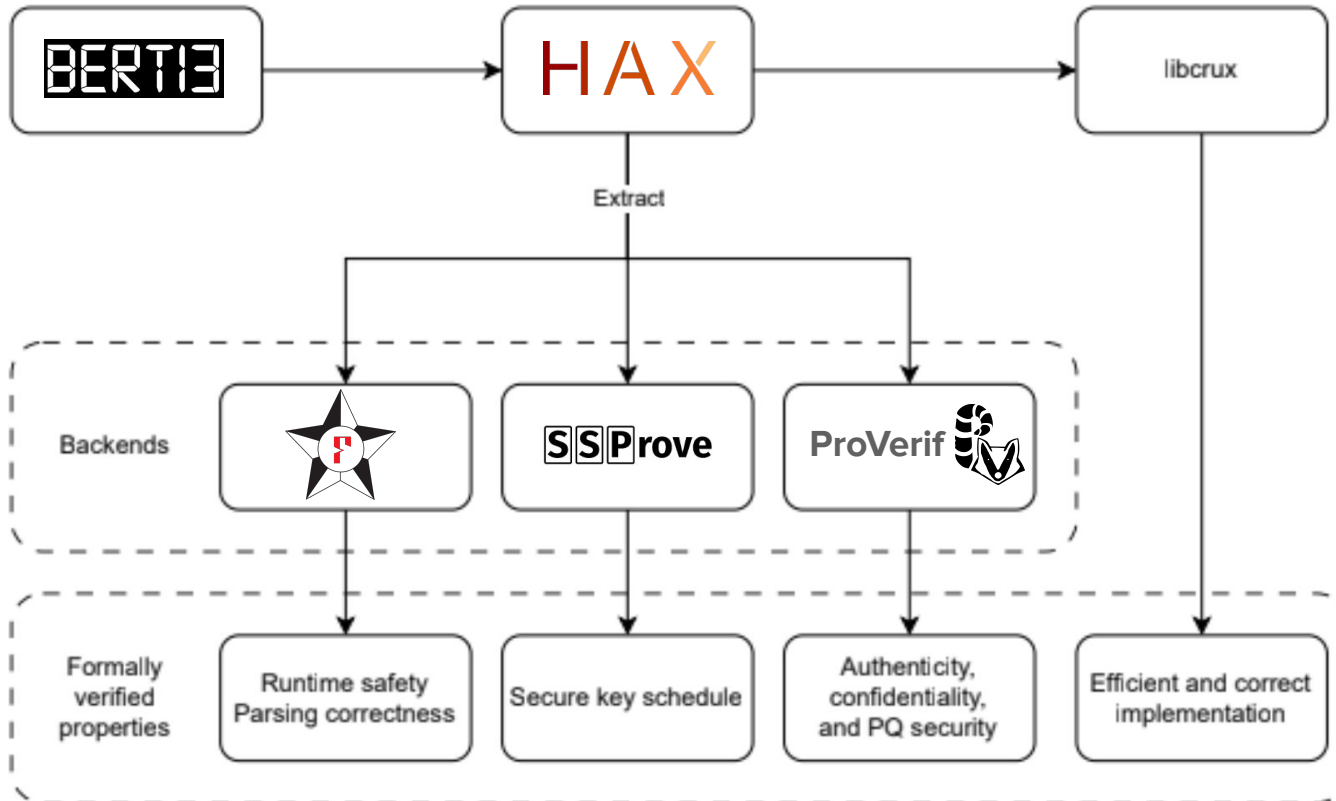
- Different properties require different proof techniques
- Memory Safety for generated C → **Low***
- Cryptographic Security → **State-Separating Proofs**
- Parsing Correctness → **EverParse, Comparse**



Bertie: Verified Post-Quantum TLS 1.3 in Rust



Verifying Bertie using hax with multiple provers



Verifying Bertie using hax with multiple provers

- Rust type system to enforce state machine invariants
- libcrux (hac + F^*) for verified cryptography
- F^* for properties like parsing correctness and panic-freedom
- ProVerif to prove protocol security for the TLS 1.3 implementation
- 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

Developer-driven Verification for Modern Crypto Protocols

CRYSPEN

Speak Freely

Say "hello" to a different messaging experience. An unexpected focus on privacy, combined with all of the features you expect.

[Get Signal](#)



Google Docs Off

Why use Signal?

Explore below to see why Signal is a simple, powerful, and secure messenger

The (Classical) Signal Protocol

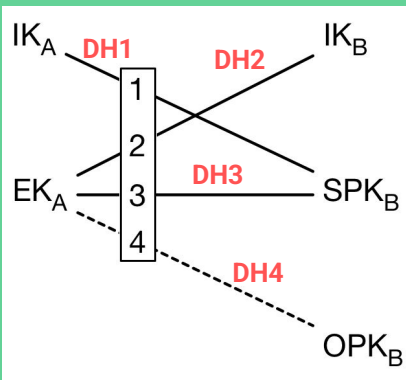
Two parts:

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

Important security guarantees:

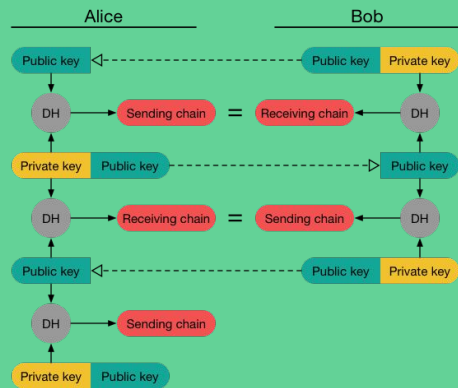
- Mutual authentication
- Post-compromise security
- Forward secrecy
- Deniability

X3DH

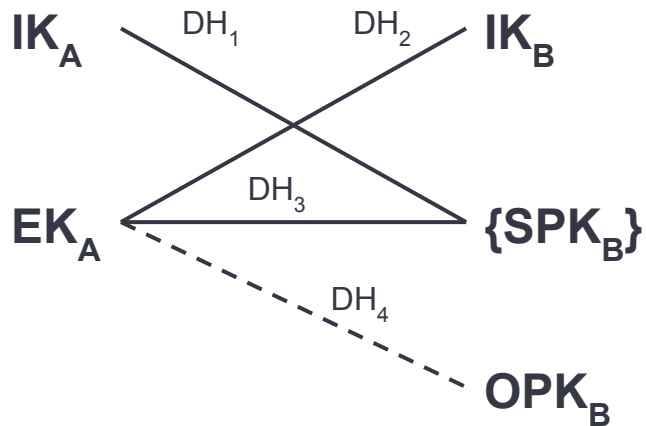


$$SK = KDF(DH1 \parallel DH2 \parallel DH3 \parallel DH4)$$

Double Ratchet

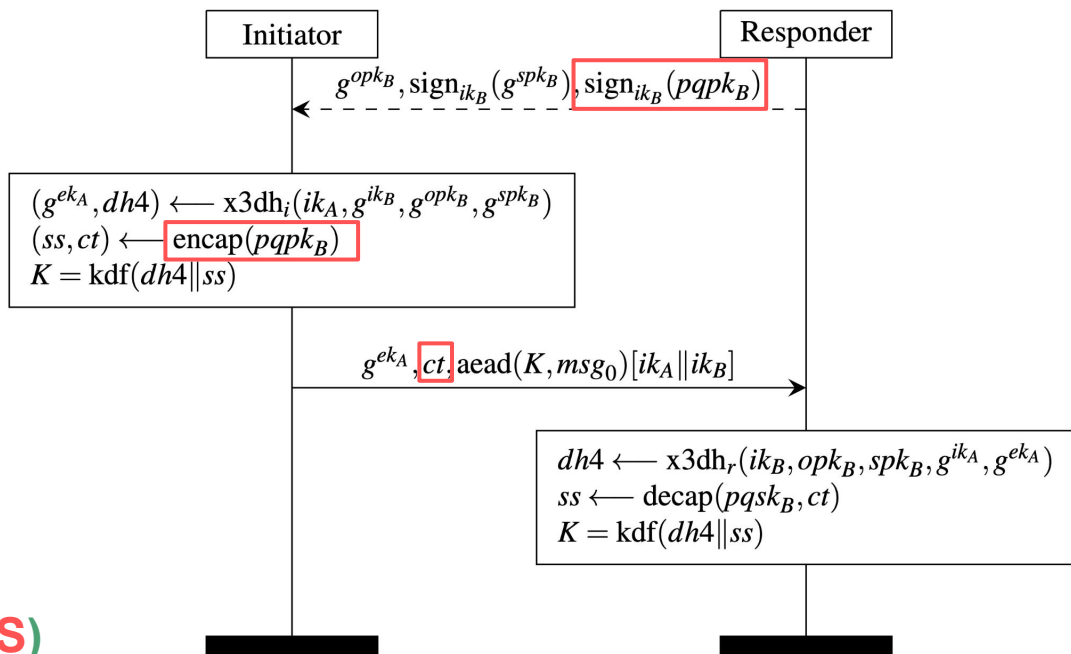


PQXDH Design: Add a PQ-KEM to X3DH



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

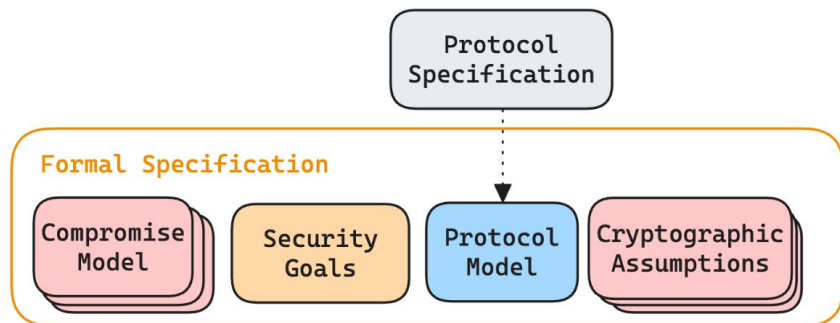
$SK = KDF(DH_1 \parallel DH_2 \parallel DH_3 \parallel DH_4 \parallel SS)$



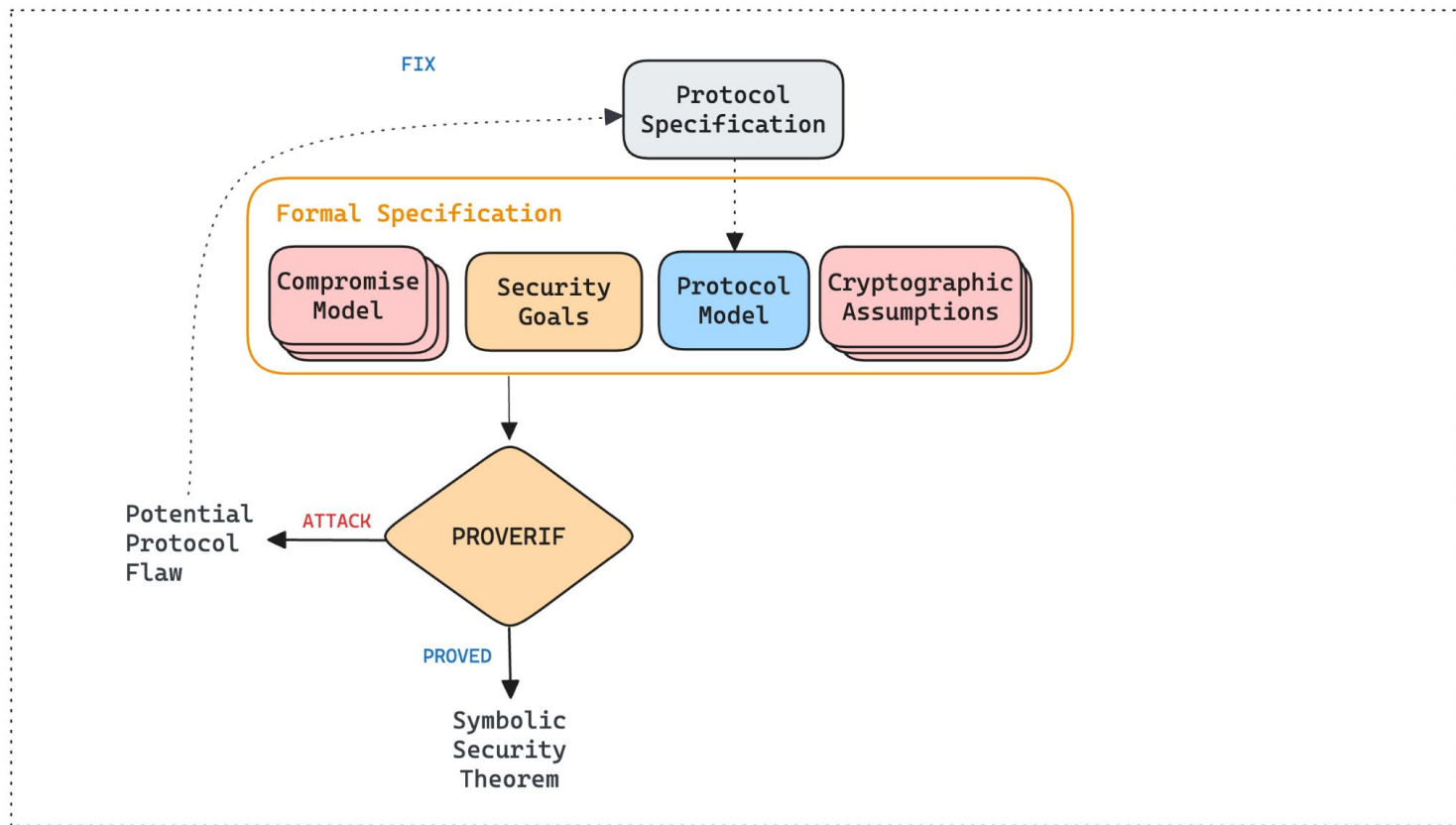
Analyzing PQXDH

- 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?

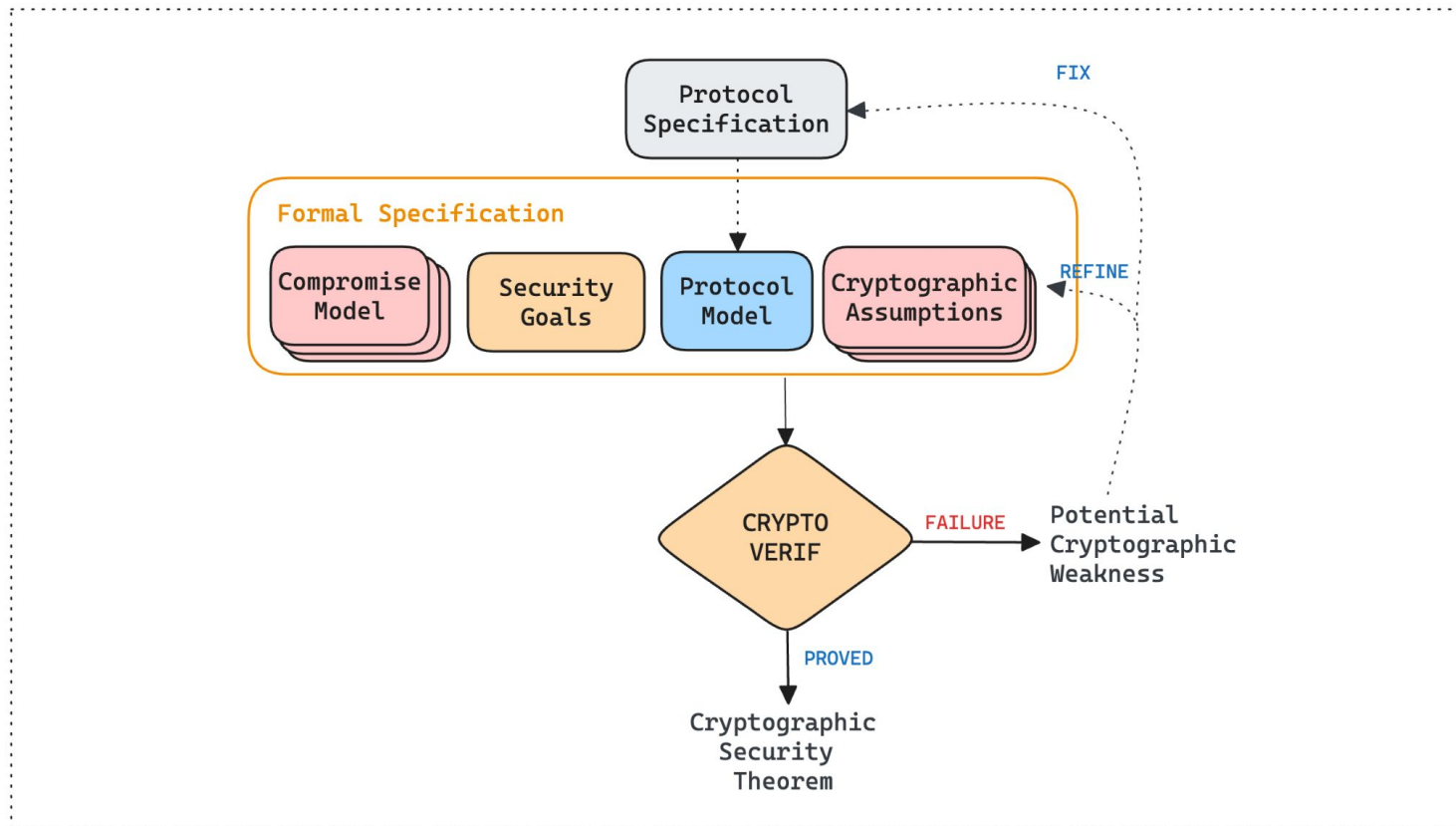
Our Formal Verification Methodology



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

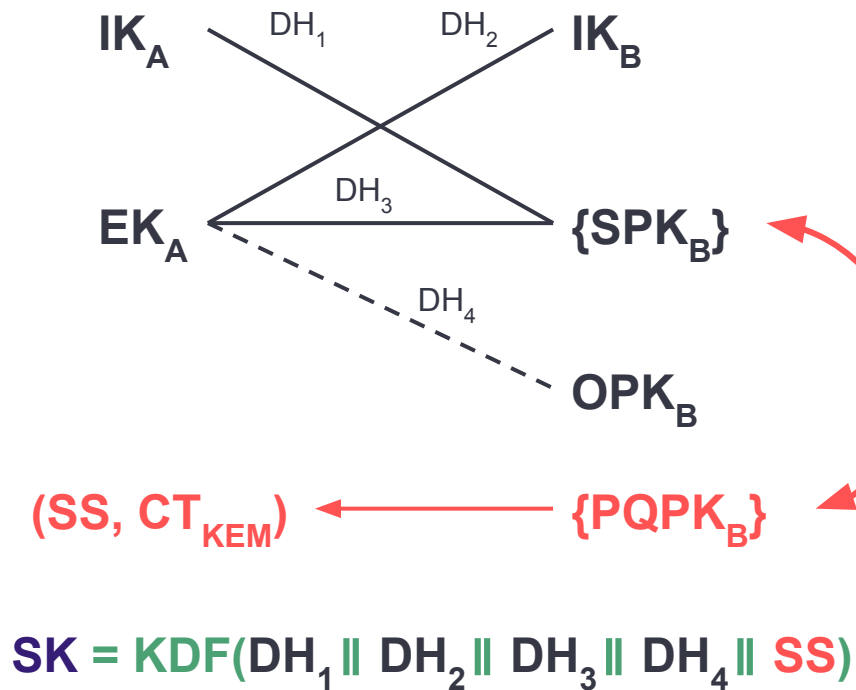
Passive Quantum Adversary Model (Harvest-Now-Decrypt-Later)

- 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

Representative of a general class of **cross-protocol attacks** between different crypto modes in the same protocol.

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

Signal implementation already does this

Long-term Fix: Use a formally verified parsing framework like Comparsé [Wallez et al., ACM CCS 2023]

PQXDH Revision and Security Theorems

Verification results led to a **new revision** of the PQXDH protocol spec:

- 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 prove that PQXDH meets its classical and PQ security requirements in the symbolic, computational, and HNDL quantum adversary models.

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

Verifying SPQR from Design to Deployment

Formally verified PQ protocol design

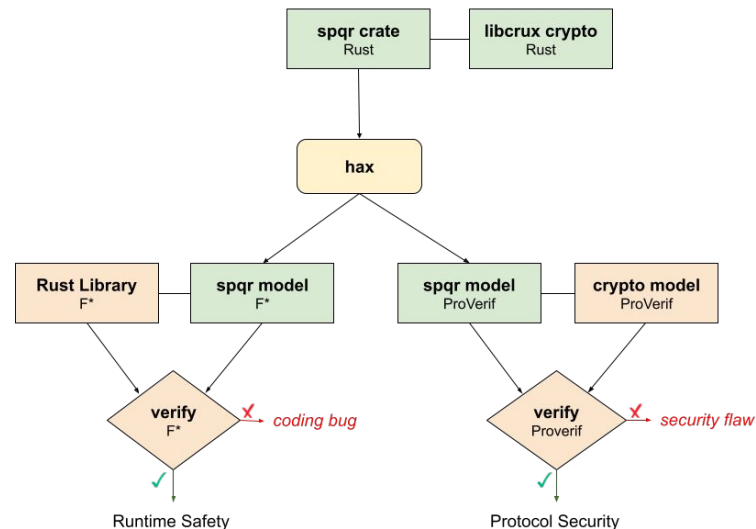
- Designers used ProVerif themselves
- Many variants analyzed and compared

Software verification for deployed code

- Verified crypto from libcrux
- Proofs of panic-freedom using hax+F*
- Security proofs using hax+ProVerif

Integrated into Signal's development

- Developers write annotations in Rust
- Developers fix simple proof breakages
- Proofs integrated in Signal's CI



Lessons and Limitations

Choice of programming and spec language is key for adoption

- Rust + annotations is more accessible to developers than F^* or C

Using multiple task-specific provers lowers verification effort

- ProVerif, F^* , SSProve have completely different strengths

... but we do not get an end-to-end security theorem

- No formal link between ProVerif, F^* , SSProve
- Weaker computational security guarantees than (say) miTLS, but over a larger part of the protocol

Conclusion

Formal verification has had a big impact on real-world cryptography

- Verified crypto libraries in all major web browsers
- Verified designs and implementations for TLS, MLS, PQ3, Signal....

The next challenge is to make verification usable by crypto developers

- Languages like Rust and Jasmin can help bridge the gap
- A need for crypto-oriented compilers, side-channel verification tools

The Post-Quantum transition offers new and exciting verification projects

- Many new algorithms, protocols, optimized implementations to verify
- High impact: provably improve security for the Internet

Questions?

- *SoK: Computer-Aided Cryptography*
[Barbosa, Barthe, Bhargavan, Blanchet, Cremers, Liao, Parno, IEEE S&P 2021]
- *Project Everest: Perspectives from Developing Industrial-Grade High-Assurance Software*
<https://project-everest.github.io/assets/everest-perspectives-2025.pdf>
- Recent Papers: <https://cryspen.com/research/>
- Code:
 - Hax: <https://github.com/cryspen/hax>
 - Libcrux: <https://github.com/cryspen/libcrux>
 - Bertie: <https://github.com/cryspen/bertie>
-

CRYSPEN

<https://cryspen.com>