Building a Formally Verified High-Performance Multi-Platform Cryptographic Library in F*

Presented by: Marina Polubelova

January 17, 2022

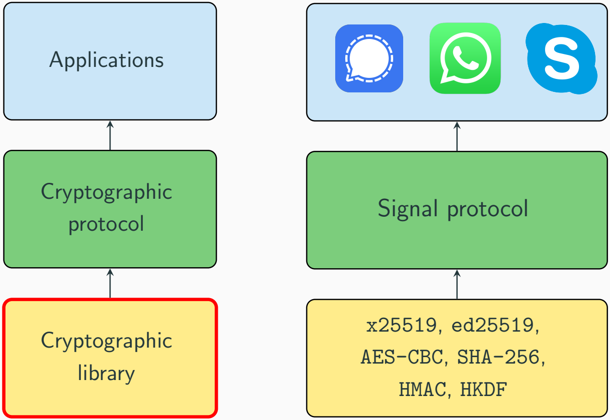
Prosecco, INRIA Paris

Jury Members

President:	Véronique CORTIER	Reviewers:	Peter SCHWABE
			Paul ZIMMERMANN
Supervisor:	Karthikeyan BHARGAVAN	Examiners:	Manuel BARBOSA
			Jonathan PROTZENKO

High-Performance Cryptography

Many security-critical applications need **efficient** and **secure** implementations of cryptographic algorithms



Multi-Platform Cryptographic Library

To address the demand for high-performance crypto, general-purpose libraries include dozens of mixed assembly-C implementations for each primitive, highly optimized for multiple platforms

- e.g., OpenSSL includes 14 implementations for Poly1305

File	LoC
poly1305-x86_64.pl	3287
poly1305-ppc.pl	1620
poly1305-x86.pl	1411
poly1305-armv4.pl	998
poly1305-sparcv9.pl	886
poly1305-s390x.pl	755
poly1305-armv8.pl	747
poly1305-ppcfp.pl	614

File	LoC
poly1305.c	333
poly1305_ieee754.c	320
poly1305-mips.pl	318
poly1305-ia64.S	302
poly1305-c64xplus.pl	269
poly1305_base2_44.c	115
poly1305_ppc.c	37
Total	12012

High-Assurance Cryptography

It is notoriously hard to write cryptographic code that is
fast, *secure* and *functionally correct*

CVE	Vulnerability	Broken property
2018-5407	EC multiplication timing leak	side-channel resistance
2018-0734	bignum timing leak	side-channel resistance
2018-0737	bignum timing leak	side-channel resistance
2017-3736	carry propagation bug	functional correctness
2017-3732	carry propagation bug	functional correctness
2017-3731	out of bounds access	memory safety
2016-7054	incorrect memset	memory safety
2016-6303	integer overflow	functional correctness
...	...	...

Formal Verification

Testing and fuzzing might help find some bugs, but not all

This work advocates the use of **formal verification** to *mathematically prove* the absence of such implementation bugs

- **Proof assistants**: Coq, F⋆, Why3, Idris, Agda, etc.
- **Prior Research Projects**

  - FIAT-crypto (2019)
  - HACL⋆ (2017)

  → C code —— Assembly code ←

  - Jasmin (2017)
  - Vale (2017)

  ↑

  - CryptoLine (2017)
  - SAW/Cryptol (2016)

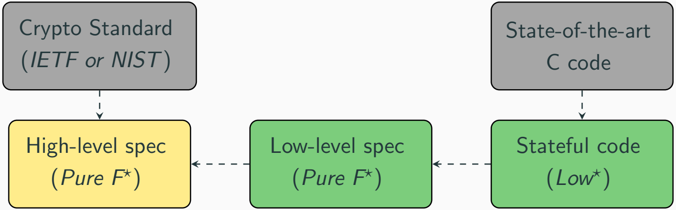
HACL*: a verified C cryptographic library (2017)

- A library of verified cryptographic algorithms
  - AEAD: ChaCha20-Poly1305
  - ECC: Curve25519, Ed25519
  - Hashes: SHA-256 and SHA-512
  - HMAC and HKDF
  - High-level APIs: `crypto_box` and `crypto_secretbox`
- Developed as a collaboration between the Prosecco team at INRIA Paris and Microsoft Research

HACL*: a verified C cryptographic library (2017)

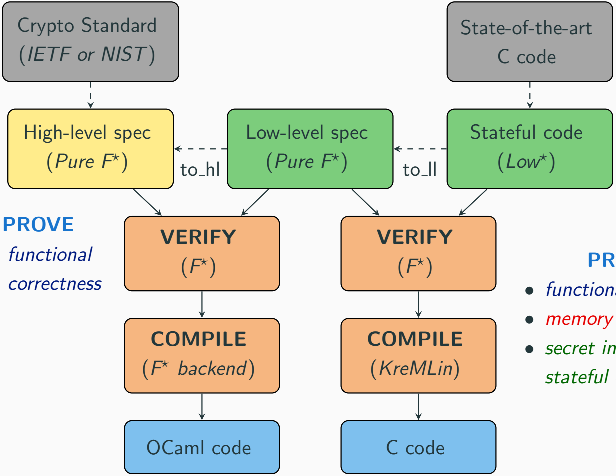
- Implemented and verified in F* and compiled to C
  - **Memory safety** proved in the C memory model
  - **Secret independence** ("constant-time") enforced by typing
  - **Functional correctness** against a mathematical specification
- Generates readable, portable, standalone C code
  - Performance comparable to hand-written C crypto libraries
  - Used in Mozilla Firefox, Wireguard VPN, miTLS, etc.

HACL* Programming workflow



- **High-level spec:** a mathematical spec of a crypto primitive
- **Low-level spec:** a pure spec of an *optimized* algorithm
- **Stateful code:** a Low* impl of the optimized algorithm

HACL⋆ Programming and Verification workflow



PROVE

- *functional correctness*

PROVE

- *functional correctness*
- *memory safety*
- *secret independence of stateful code*

Writing Verified Cryptographic Code (Curve25519)

The Curve25519 elliptic curve is standardized as **IETF RFC7748**

Crypto Standard
(*IETF or NIST*)

Algorithm
Pseudo code

High-level spec
(*Pure F**)

State-of-the-art
C code

Stateful code
(*Low**)

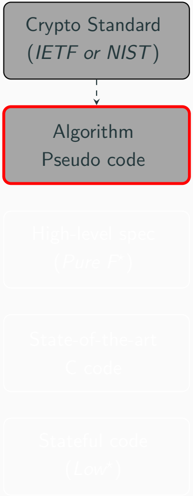
```
Internet Research Task Force (IRTF)                           A. Langley
Request for Comments: 7748                                        Google
Category: Informational                                       M. Hamburg
ISSN: 2070-1721                             Rambus Cryptography Research
                                                               S. Turner
                                                                   sn3rd
                                                            January 2016


                    Elliptic Curves for Security

Abstract

   This memo specifies two elliptic curves over prime fields that offer
   a high level of practical security in cryptographic applications,
   including Transport Layer Security (TLS).  These curves are intended
   to operate at the ~128-bit and ~224-bit security level, respectively,
   and are generated deterministically based on a list of required
   properties.
```

Writing Verified Cryptographic Code (Curve25519)



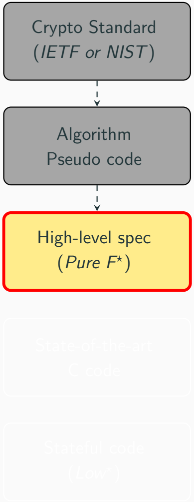

```
For t = bits-1 down to 0:
    k_t = (k >> t) & 1
    swap ^= k_t
    // Conditional swap; see text below.
    (x_2, x_3) = cswap(swap, x_2, x_3)
    (z_2, z_3) = cswap(swap, z_2, z_3)
    swap = k_t

    A = x_2 + z_2
    AA = A^2
    B = x_2 - z_2
    BB = B^2
    E = AA - BB
    C = x_3 + z_3
    D = x_3 - z_3
    DA = D * A
    CB = C * B
    x_3 = (DA + CB)^2
    z_3 = x_1 * (DA - CB)^2
    x_2 = AA * BB
    z_2 = E * (AA + a24 * E)

// Conditional swap; see text below.
(x_2, x_3) = cswap(swap, x_2, x_3)
(z_2, z_3) = cswap(swap, z_2, z_3)
Return x_2 * (z_2^(p - 2))
```

Writing Verified Cryptographic Code (Curve25519)

High-level spec uses mathematical operations over arbitrary size integers




```
let prime : pos = pow2 255 - 19
let elem = x:ℕ{x < prime}

let ( +% ) (x y:elem) : elem = (x + y) % prime
let ( -% ) (x y:elem) : elem = (x - y) % prime
let ( *% ) (x y:elem) : elem = (x × y) % prime

let add_and_double (x_1,z_1) (x_2,z_2) (x_3,z_3) =
  let a = x_2 +% z_2 in
  let aa = a *% a in
  let b = x_2 -% z_2 in
  let bb = b *% b in
  let e = aa -% bb in
  let c = x_3 +% z_3 in
  let d = x_3 -% z_3 in
  let da = d *% a in
  let cb = c *% b in
  let x_3 = (da +% cb) *% (da +% cb) in
  let z_3 = x_1 *% (da -% cb) *% (da -% cb) in
  let x_2 = aa *% bb in
  let z_2 = e *% (aa +% 121665 *% e) in
  (x_2,z_2), (x_3,z_3)
```

Writing Verified Cryptographic Code (Curve25519)

State-of-the-art C code: Adam Langley's `donna-c64` is the portable C implementation of Curve25519 for 64-bit platforms

Crypto Standard
(*IETF or NIST*)

Algorithm
Pseudo code

High-level spec
(Pure $F^\star$)

State-of-the-art
C code

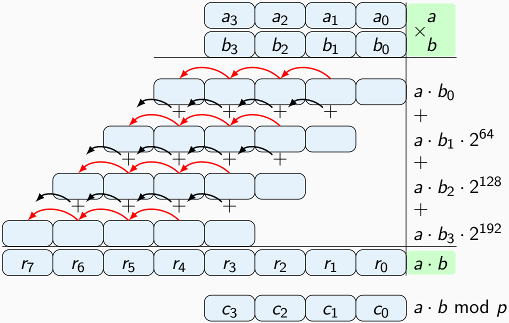
Stateful code
($Low^\star$)

```
static void
fmonty(limb *x2, limb *z2, /* output 2Q */
       limb *x3, limb *z3, /* output Q + Q' */
       limb *x, limb *z,   /* input Q */
       limb *xprime, limb *zprime, /* input Q' */
       const limb *qmqp /* input Q - Q' */) {
  limb origx[5], origxprime[5], zzz[5], xx[5], zz[5], xxprime[5],
        zzprime[5], zzzprime[5];

  memcpy(origx, x, 5 * sizeof(limb));
  fsum(x, z);
  fdifference_backwards(z, origx);  // does x - z

  memcpy(origxprime, xprime, sizeof(limb) * 5);
  fsum(xprime, zprime);
  fdifference_backwards(zprime, origxprime);
  fmul(xxprime, xprime, z);
  fmul(zzprime, x, zprime);
  memcpy(origxprime, xxprime, sizeof(limb) * 5);
  fsum(xxprime, zzprime);
  fdifference_backwards(zzprime, origxprime);
  fsquare_times(x3, xxprime, 1);
  fsquare_times(zzzprime, zzprime, 1);
  fmul(z3, zzzprime, qmqp);
```

Modular Multiplication for Curve25519 on 64-bit platforms

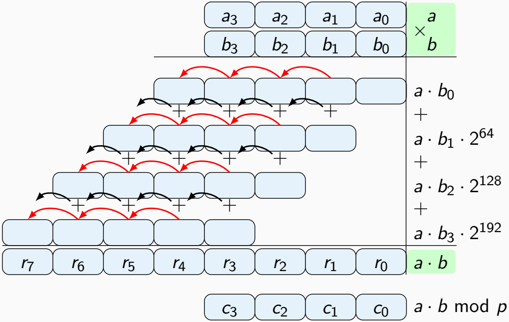


- $p = 2^{255} - 19$, each field element has up to 255 bits
- Field arithmetic with a radix-2^{64} representation

  $a = a_0 + a_1 \cdot 2^{64} + a_2 \cdot 2^{128} + a_3 \cdot 2^{192}$

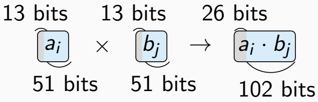
  a is stored as an array of four 64-bit unsigned integers
- Modular reduction: $2^{256} \bmod p = 38$

What can go wrong?



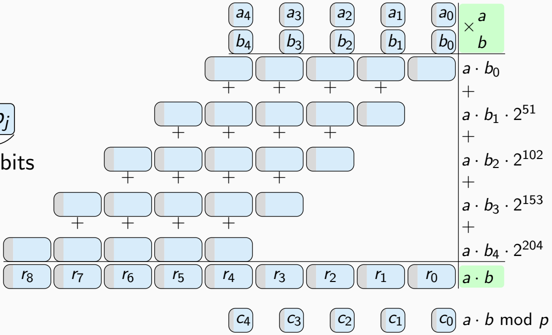
- **Functional Correctness:** missing carry propagation steps?
- **Memory Safety:** accessing arrays a and b out of bounds?
- **Secret Independence:** skipping multiplications with zero?

Faster Modular Multiplication for Curve25519



no carry propagation

until mod p

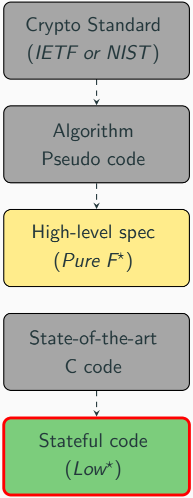


- Multiplication in radix-2^{64} is too slow
- The well-known optimization is to use radix-2^{51}

  $a = a_0 + a_1 \cdot 2^{51} + a_2 \cdot 2^{102} + a_3 \cdot 2^{153} + a_4 \cdot 2^{204}$

  a is stored as an array of five 64-bit unsigned integers
- Modular reduction: $2^{255} \bmod p = 19$
- Implemented in `donna-c64`, `fiat-crypto`, HACL*, etc.

Writing Verified Cryptographic Code (Curve25519)




```
let felem5 = lbuffer sec_uint64 5ul

val fmul (out f1 f2:felem5) : Stack unit
  (requires λ h →
    live h out ∧ live h f1 ∧ live h f2 ∧
    eq_or_disjoint f1 f2 ∧ eq_or_disjoint f1 out ∧
    eq_or_disjoint f2 out ∧ fmul_pre h f1 f2)
  (ensures λ h0 _ h1 →
    modifies (loc out) h0 h1 ∧ fmul_post h1 out ∧
    feval5 h1 out == feval5 h0 f1 *% feval5 h0 f2)
```

- Memory Safety
- Functional Correctness
- Secret Independence

Secret Independence

```
(* the type of secret integers is abstract *)
val sec_int_t: inttype → Type0

let int_t (t:inttype) (l:secrecy_level) = match (l, t) with
  | SEC, _ → sec_int_t t
  | PUB, U8  → LowStar.UInt8.t | ...

val logand: #t:inttype → #l:secrecy_level
  → int_t t l → int_t t l → int_t t l

val lt: #t:inttype → int_t t PUB → int_t t PUB → bool
```

- Define the set of constant-time operations on secret integers
    - Constant-time operations: +, *, -, ^, &, |, ~, >>, <<
    - Variable-time operations: /, %, ==, <, >
    - Depends on the target platform
- Secret integers **cannot** be used for branching, array indices, array lengths, and loop counters

Open Problems: Performance

There is a significant gap in performance between verified C code and assembly ($1.1 - 5.7\times$)

How can we bridge this gap?

- Can we write verified assembly for each platform?

  It seems hard

- **Our approach:** obtain verified optimized code for multiple platforms from one *generic* implementation in F*

- FIAT-crypto (2019)
- HACL* (2017)

$\longrightarrow$ C code ——— Assembly code $\longleftarrow$

- Jasmin (2017)
- Vale (2017)

$\uparrow$

- CryptoLine (2017)
- SAW/Cryptol (2016)

Open Problems: Arbitrary-Precision Arithmetic

There is no verified implementation of cryptographic algorithms that rely on arbitrary-precision arithmetic

Can we implement and verify such algorithms in F*?

- a *constant-time* bignum library
- a *portable* bignum library
- an implementation of RSA-PSS and FFDHE (2048 – 8192 bits)
  needed for signing and key exchange in TLS 1.3
- Bignum256, Bignum512, Bignum4096, etc.
  needed for elliptic curves and ElectionGuard

Our approach and results

We write *generic* verified code in F* that compiles to optimized C code for different platforms, composable with verified assembly

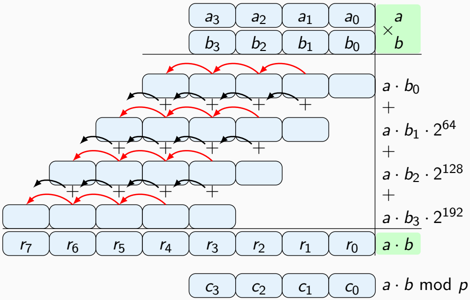
- **EverCrypt: a Verified Cryptographic Provider**
  - share the code between assembly and C implementations
  - Curve25519
- **A Verified Bignum Library**
  - share the code between 32-bit and 64-bit bignum libraries
  - RSA-PSS, FFDHE, Ed25519
- **HACL$\times$N: Verified Generic SIMD Crypto**
  - share the code between scalar and vectorized implementations
  - ChaCha20-Poly1305, SHA2-mb, Blake2

Our approach and results

We write *generic* verified code in $F^\star$ that compiles to optimized C code for different platforms, composable with verified assembly

- **EverCrypt: a Verified Cryptographic Provider**
  - share the code between assembly and C implementations
  - Curve25519

Faster Modular Multiplication for Curve25519



Radix-2^{64} multiplication can be implemented efficiently using the Intel `ADX` and `MULX` instructions

- Two addition instructions `ADOX` and `ADCX` compute addition with a carry using two independent carry flags
- We can implement multiplication with two parallel carry chains
- These instructions are not available in C, so we have to write this function in assembly

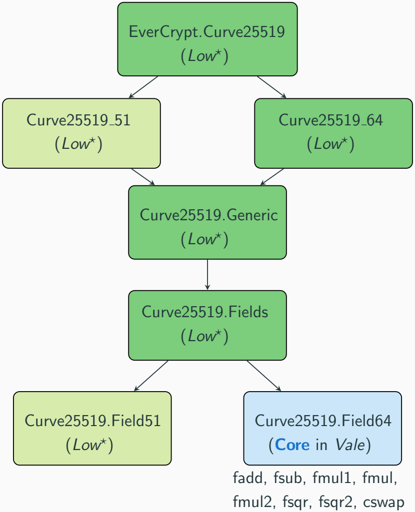
Mixed assembly-C implementation of Curve25519

HACL⋆-v1

- Verified a radix-2^{51} monolithic implementation

HACL⋆-v2

- Completely restructured the code to allow multiple field arithmetic implementations
- Identified performance critical functions for radix-2^{64}
- The Vale project implemented and verified them in Intel assembly
- Verified the composition of Low⋆ and Vale code in F⋆



Multiplexing for Modular Multiplication

```
type field_spec = | M51 | M64

let felem (s:field_spec) = match s with
  | M51 → lbuffer sec_uint64 5ul
  | M64 → lbuffer sec_uint64 4ul

let fmul (#s:field_spec) (out f1 f2:felem s) : Stack unit
  (requires λ h →
    live h out ∧ live h f1 ∧ live h f2 ∧
    eq_or_disjoint f1 f2 ∧ eq_or_disjoint f1 out ∧
    eq_or_disjoint f2 out ∧ fmul_pre h f1 f2)
  (ensures λ h0 _ h1 →
    modifies (loc out) h0 h1 ∧ fmul_post h1 out ∧
    feval h1 out == feval h0 f1 *% feval h0 f2) =

  match s with
  | M51 → fmul_51 out f1 f2 (* from Low* implementation *)
  | M64 → fmul_64 out f1 f2 (* from Vale implementation *)
```

- Memory Safety, Functional Correctness, Secret Independence
- Multiplexing: composing multiple field arithmetic implementations

Curve25519 Performance

Implementation	Radix	Language	CPU cycles
donna-c64	2^{51}	64-bit C	159634
fiat-crypto	2^{51}	64-bit C	145248
amd64-64	2^{51}	Intel x86_64 asm	143302
sandy2x	$2^{25.5}$	Intel AVX asm	135660
HACL*-v2 portable	2^{51}	64-bit C	**135636**
openssl	2^{64}	Intel ADX asm	118604
Oliveira et al.	2^{64}	Intel ADX asm	115122
HACL*-v2 targeted	2^{64}	64-bit C	**113614**
		+ Intel ADX asm	

Our approach and results

We write *generic* verified code in $F^\star$ that compiles to optimized C code for different platforms, composable with verified assembly

- **EverCrypt: a Verified Cryptographic Provider**
  - share the code between assembly and C implementations
  - Curve25519
- **A Verified Bignum Library**
  - share the code between 32-bit and 64-bit bignum libraries
  - RSA-PSS, FFDHE, Ed25519
- **HACL$\times$N: Verified Generic SIMD Crypto**
  - share the code between scalar and vectorized implementations
  - ChaCha20-Poly1305, SHA2-mb, Blake2

Our approach and results

We write *generic* verified code in F* that compiles to optimized C code for different platforms, composable with verified assembly

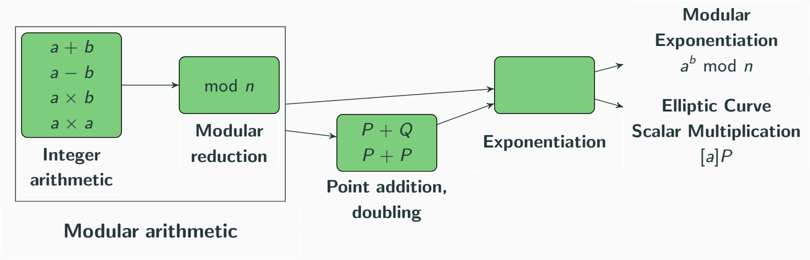
- **A Verified Bignum Library**
  - share the code between 32-bit and 64-bit bignum libraries
  - RSA-PSS, FFDHE, Ed25519

A Bignum Library

Many cryptographic algorithms work with large numbers that do not fit within a machine word

- Elliptic Curve Cryptography
  - arithmetic modulo a prime of several *hundred* bits in size
  - Curve25519, Curve448, P-256, P-384, P-521, etc.
  - a modulus is usually known *in advance*
  - a "default" implementation for *any* prime of *any* size
- Finite-Field Cryptography
  - arithmetic modulo a large number of *thousands* bits in size
  - RSA, RSA-PSS, FFDHE, ElGamal, Paillier, etc.
  - a modulus is *not known in advance*, it is not always a prime, even its size is unknown

A Verified Bignum Library



Exponentiation is defined as a repeated application of a commutative monoid operation

- *Modular Exponentiation*: repeated modular multiplication
- *Elliptic Curve Scalar Multiplication*: repeated point addition

Modular Exponentiation

$$a^b \bmod n = \underbrace{a \cdot a \cdot \ldots \cdot a}_{b \text{ times}} \bmod n$$

Modular Exponentiation

$$a^b \bmod n = \underbrace{a \cdot a \cdot \ldots \cdot a}_{b \text{ times}} \bmod n$$

$$a^b \bmod n = (\ldots((a \cdot a) \bmod n \cdot a) \bmod n \cdot \ldots \cdot a) \bmod n$$

The naive method requires $b - 1$ modular multiplications!

Modular Exponentiation

$$a^b \bmod n = \underbrace{a \cdot a \cdot \ldots \cdot a}_{b \text{ times}} \bmod n$$

$$a^b \bmod n = (\ldots((a \cdot a) \bmod n \cdot a) \bmod n \cdot \ldots \cdot a) \bmod n$$

- **Generic methods**, where a and b may vary
  - Binary method, Fixed-window method
- *Fixed base methods*, where a is fixed
  - Fixed-base comb method
- *Fixed exponent methods*, where b is fixed
  - Addition-chain method

Binary Method for Modular Exponentiation

$$a^b \bmod n = \underbrace{a \cdot a \cdot \ldots \cdot a}_{b \text{ times}} \bmod n$$

- a binary representation for an exponent b:
  $$a^b = a^{(b_\ell \ldots b_2 b_1 b_0)_2} = a^{b_\ell \cdot 2^\ell + \ldots + b_2 \cdot 2^2 + b_1 \cdot 2 + b_0}$$

Binary Method for Modular Exponentiation

$$a^b \bmod n = \underbrace{a \cdot a \cdot \ldots \cdot a}_{b \text{ times}} \bmod n$$

- a binary representation for an exponent b:
$$a^b = a^{(b_\ell \ldots b_2 b_1 b_0)_2} = a^{b_\ell \cdot 2^\ell + \ldots + b_2 \cdot 2^2 + b_1 \cdot 2 + b_0}$$
- using Horner's method we can write it as follows
$$a^b = (((\ldots(1^2 \cdot a^{b_\ell})^2 \ldots)^2 \cdot a^{b_2})^2 \cdot a^{b_1})^2 \cdot a^{b_0}$$

Binary Method for Modular Exponentiation

$$a^b \bmod n = \underbrace{a \cdot a \cdot \ldots \cdot a}_{b \text{ times}} \bmod n$$

- a binary representation for an exponent b:
$$a^b = a^{(b_\ell \ldots b_2 b_1 b_0)_2} = a^{b_\ell \cdot 2^\ell + \ldots + b_2 \cdot 2^2 + b_1 \cdot 2 + b_0}$$
- using Horner's method we can write it as follows
$$a^b = (((\ldots(1^2 \cdot a^{b_\ell})^2 \ldots)^2 \cdot a^{b_2})^2 \cdot a^{b_1})^2 \cdot a^{b_0}$$
- **Left-to-right binary method**

$$acc_i = a^{(b_\ell \ldots b_{\ell-i})_2}$$
$$= ((\ldots(1^2 \cdot a^{b_\ell})^2 \ldots)^2 \cdot a^{b_{\ell-(i-1)}})^2 \cdot a^{b_{\ell-i}}$$
$$= (acc_{i-1})^2 \cdot a^{b_{\ell-i}}$$

$$(b_\ell \ldots \underbrace{b_{\ell-i}}_{1 \text{ bit}} \ldots b_2 b_1 b_0)_2$$

Fixed-Window Method for Modular Exponentiation

$$a^b \bmod n = \underbrace{a \cdot a \cdot \ldots \cdot a}_{b \text{ times}} \bmod n$$

- a radix-2^w representation for an exponent b:

$$a^b = a^{(b_\ell \ldots b_2 b_1 b_0)_{2^w}} = a^{b_\ell \cdot (2^w)^\ell + \ldots + b_2 \cdot (2^w)^2 + b_1 \cdot (2^w) + b_0}$$

$$a^b = (((\ldots(1^{2^w} \cdot a^{b_\ell})^{2^w} \ldots)^{2^w} \cdot a^{b_2})^{2^w} \cdot a^{b_1})^{2^w} \cdot a^{b_0}$$

- **Left-to-right fixed-window method**

$$acc_i = a^{(b_\ell \ldots b_{\ell-i})_{2^w}}$$

$$= ((\ldots(1^{2^w} \cdot a^{b_\ell})^{2^w} \ldots)^{2^w} \cdot a^{b_{\ell-(i-1)}})^{2^w} \cdot a^{b_{\ell-i}}$$

$$= (acc_{i-1})^{2^w} \cdot a^{b_{\ell-i}}$$

$$(b_\ell \ldots \underbrace{b_{\ell-i}}_{w \text{ bits}} \ldots b_2 b_1 b_0)_{2^w}$$

Verified Exponentiation

```
class comm_monoid (t:Type) = {
  one: t;
  mul: t → t → t;
  lemma_one: a:t → Lemma (mul a one == a);
  lemma_mul_assoc: a:t → b:t → c:t →
    Lemma (mul (mul a b) c == mul a (mul b c));
  lemma_mul_comm: a:t → b:t → Lemma (mul a b == mul b a)
}

val exp_l2r_lemma: #t:Type → k:comm_monoid t
  → a:t → bBits:ℕ → b:ℕ{b < pow2 bBits} →
  Lemma (exp_l2r k a bBits b == pow k a b)
```

- Functional Correctness: Left-to-right binary method matches a mathematical definition of exponentiation

Squaring

How to compute $a \cdot a$ efficiently?

Squaring

How to compute $a \cdot a$ efficiently?

$$
\begin{array}{cccccc|l}
 & a_\ell & \dots & a_2 & a_1 & a_0 & a \\
 & a_\ell & \dots & a_2 & a_1 & a_0 & \times\, a \\
\hline
 & & & a \cdot a_0 & & & a \cdot a_0 \\
 & & a \cdot a_1 & & & & + \; a \cdot a_1 \cdot \beta \\
 & a \cdot a_2 & & & & & + \; a \cdot a_2 \cdot \beta^2 \\
 & \dots & & & & & \dots \\
a \cdot a_\ell & & & & & & + \; a \cdot a_\ell \cdot \beta^\ell \\
\hline
r_{2\cdot\ell+1} & \dots & & r_2 & r_1 & r_0 & res
\end{array}
$$

$$
\begin{array}{c|l}
a_\ell \quad \dots \quad a_2 \quad a_1 \quad a_0 & a \\
a_\ell \quad \dots \quad a_2 \quad a_1 \quad a_0 & \times\, a \\
\hline
a_0 \cdot a_\ell \cdot \beta^\ell + \dots + a_0 \cdot a_2 \cdot \beta^2 + a_0 \cdot a_1 \cdot \beta^1 + a_0 \cdot a_0 \cdot \beta^0 & a \cdot a_0 \\
a_1 \cdot a_\ell \cdot \beta^{\ell+1} + \dots + a_1 \cdot a_2 \cdot \beta^3 + a_1 \cdot a_1 \cdot \beta^2 + a_1 \cdot a_0 \cdot \beta^1 & + \; a \cdot a_1 \cdot \beta \\
a_2 \cdot a_\ell \cdot \beta^{\ell+2} + \dots + a_2 \cdot a_2 \cdot \beta^4 + a_2 \cdot a_1 \cdot \beta^3 + a_2 \cdot a_0 \cdot \beta^2 & + \; a \cdot a_2 \cdot \beta^2 \\
\dots & + \; \dots \\
a_\ell \cdot a_\ell \cdot \beta^{2\cdot\ell} + \dots + a_\ell \cdot a_2 \cdot \beta^{\ell+2} + a_\ell \cdot a_1 \cdot \beta^{\ell+1} + a_\ell \cdot a_0 \cdot \beta^\ell & + \; a \cdot a_\ell \cdot \beta^\ell \\
\hline
r_{2\cdot\ell+1} \quad \dots \quad r_2 \quad r_1 \quad r_0 & res
\end{array}
$$

Generic Bignum Representation and Verified Squaring

```
let limb_t = t:inttype{t = U32 ∨ t = U64}
let lbignum (t:limb_t) (len:size_t) = lbuffer (uint_t t SEC) len

val bn_sqr (#t:limb_t) (len:bn_len t) (a:lbignum t len)
  (res:lbignum t (len +! len)) : Stack unit
  (requires λ h →
    live h a ∧ live h res ∧ disjoint res a)
  (ensures  λ h0 _ h1 →
    modifies (loc res) h0 h1 ∧
    lbn_v h1 res == lbn_v h0 a × lbn_v h0 a)
```

- Memory Safety
- Functional Correctness
- Secret Independence

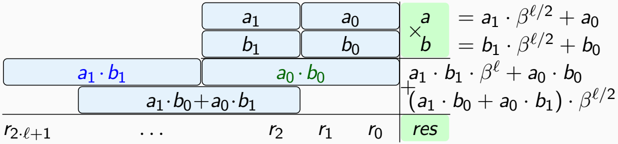
Karatsuba Multiplication

How to compute $a \cdot b$ efficiently?

Karatsuba Multiplication

How to compute $a \cdot b$ efficiently?

The well-known optimization is Karatsuba Multiplication



$$a \cdot b = (a_1 \cdot \beta^{\ell/2} + a_0) \cdot (b_1 \cdot \beta^{\ell/2} + b_0)$$
$$= a_1 \cdot b_1 \cdot \beta^{\ell} + (a_1 \cdot b_0 + a_0 \cdot b_1) \cdot \beta^{\ell/2} + a_0 \cdot b_0$$

- **Subtractive variant**

$$a_1 \cdot b_0 + a_0 \cdot b_1 = a_1 \cdot b_1 + a_0 \cdot b_0 - (a_0 - a_1) \cdot (b_0 - b_1)$$

- **Additive variant**

$$a_1 \cdot b_0 + a_0 \cdot b_1 = (a_0 + a_1) \cdot (b_0 + b_1) - a_1 \cdot b_1 - a_0 \cdot b_0$$

Montgomery Multiplication

How to compute $a \cdot b \bmod n$ efficiently?

Montgomery Multiplication

How to compute $a \cdot b \bmod n$ efficiently?

- Montgomery multiplication replaces the expensive division by n with a fast division by a carefully chosen r
- Modular addition, subtraction, and multiplication can be efficiently done in the Montgomery domain

$$aM = a \cdot r \bmod n \quad bM = b \cdot r \bmod n \quad c = cM \cdot r^{-1} \bmod n$$

$$\begin{tikzcd}
aM, bM \arrow[r, "\text{mont_op}"] & cM \arrow[d, "\text{from_mont}"] \\
a, b \arrow[u, "\text{to_mont}"] \arrow[r, "\text{op}"'] & c
\end{tikzcd}$$

op	mont_op
$(a + b) \bmod n$	$(aM + bM) \bmod n$
$(a - b) \bmod n$	$(aM - bM) \bmod n$
$a \cdot b \bmod n$	mont_redc $(aM \cdot bM)$

Montgomery Exponentiation

How to compute $a^b \bmod n$ efficiently?

Montgomery Exponentiation

How to compute $a^b \bmod n$ efficiently?

$$
\begin{tikzcd}
aM = a \cdot r \bmod n \arrow[r, "\text{pow\_mont}", "r \cdot d \bmod n = 1"'] & (aM \cdot d)^b \cdot r \bmod n \arrow[d, "\text{from\_mont}"] \\
a \arrow[u, "\text{to\_mont}"] \arrow[r, "\text{pow\_mod}"'] & a^b \bmod n
\end{tikzcd}
$$

- `pow_mod`: repeated modular multiplication
- `pow_mont`: repeated Montgomery multiplication

Verified Montgomery Arithmetic

```
val pow_mont_is_pow_mod (n r:pos) (d:ℤ{r × d % n = 1}) (aM:nat_mod n) (b:ℕ) :
  Lemma (pow_mont n r d aM b == pow_mod #n (aM × d % n) b × r % n)

val bn_mont_mul (#t:limb_t) (k:pbn_mont_ctx t)
  (aM bM cM:buffer (uint_t t SEC)) : Stack unit
  (requires λ h → ...)
  (ensures  λ h0 _ h1 → ... ∧ modifies (loc cM) h0 h1 ∧
    bn_mont_v h1 k cM == (bn_mont_v h0 k aM × bn_mont_v h0 k bM) % bn_mont_n h0 k)
```

- Memory Safety, Functional Correctness, Secret Independence

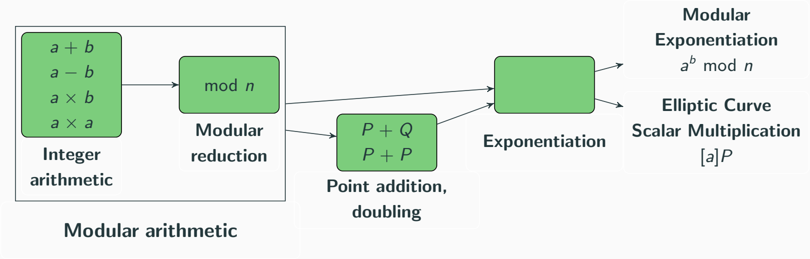
Extracted C code for modular exponentiation

```c
typedef struct bn_mont_ctx_u64_s {
  uint32_t len;
  uint64_t *n;
  uint64_t mu;
  uint64_t *r2;
} bn_mont_ctx_u64;


bn_mont_ctx_u64 *bn_mont_ctx_init(uint32_t len, uint64_t *n);

void bn_mod_exp_consttime_precomp(bn_mont_ctx_u64 *k,
  uint64_t *a, uint32_t bBits, uint64_t *b, uint64_t *res);
```

Verified Applications



- Applications of arbitrary size bignums
  - FFDHE, RSA-PSS
- Applications of fixed size bignums
  - Bignum256, Bignum4096
- Applications of exponentiation for EC Scalar Multiplication
  - Ed25519

Performance Benchmarks

Implementation	2048	3072	4096	6144	8192
`openssl-asm`	6785	21509	50414	173646	411168
`gmp-asm`	8554	27121	62724	207042	486562
`openssl-no-mulx`	10613	34773	82075	279073	670069
HACL*-v2	**15969**	**51940**	**116838**	**381314**	**878264**
`openssl-portable`	39055	113443	263119	828745	1862540
`gmp-portable`	47283	149781	425988	1442425	3388961

Performance benchmarks for constant-time modular exponentiation $a^b \bmod n$, where a, b and n are bignums of the same length. Measurements are in cycles (thousands) for input lengths ranging from 2048 to 8192 bits.

- **Future work:** use Vale code for `ADX` and `MULX` to close the performance gap

Our approach and results

We write *generic* verified code in F* that compiles to optimized C code for different platforms, composable with verified assembly

- **EverCrypt: a Verified Cryptographic Provider**
  - share the code between assembly and C implementations
  - Curve25519
- **A Verified Bignum Library**
  - share the code between 32-bit and 64-bit bignum libraries
  - RSA-PSS, FFDHE, Ed25519
- **HACL$\times$N: Verified Generic SIMD Crypto**
  - share the code between scalar and vectorized implementations
  - ChaCha20-Poly1305, SHA2-mb, Blake2

Our approach and results

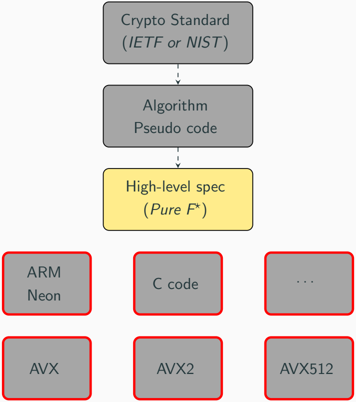
We write *generic* verified code in F* that compiles to optimized C code for different platforms, composable with verified assembly

- **HACL$\times$N: Verified Generic SIMD Crypto**
  - share the code between scalar and vectorized implementations
  - ChaCha20-Poly1305, SHA2-mb, Blake2

Multi-Platform Cryptography

How to speed up other implementations of algorithms in HACL⋆?

The biggest performance impact comes from vector instructions



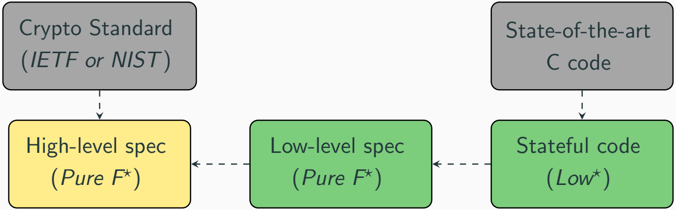
- e.g., Poly1305 in OpenSSL

File	LoC
`poly1305-x86_64.pl`	3287
`poly1305-ppc.pl`	1620
`poly1305-x86.pl`	1411
`poly1305-armv4.pl`	998
`poly1305-sparcv9.pl`	886
`poly1305-s390x.pl`	755
`poly1305-armv8.pl`	747
`poly1305-ppcfp.pl`	614
...	

HACL×N: Verified Generic SIMD Crypto

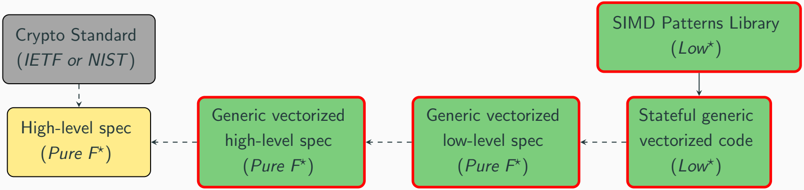
- We identify the generic SIMD crypto programming patterns:
  - Exploiting Internal Parallelism (Blake2)
  - Multiple Input Parallelism (SHA-2)
  - Counter Mode Encryption (ChaCha20)
  - Polynomial Evaluation (Poly1305)
- We write one *generic* SIMD implementation in Low* and compile it to multiple platforms:
  - 128-bit vector instructions: ARM Neon and Intel AVX
  - 256-bit vector instructions: Intel AVX2
  - 512-bit vector instructions: Intel AVX512

HACL* Programming workflow



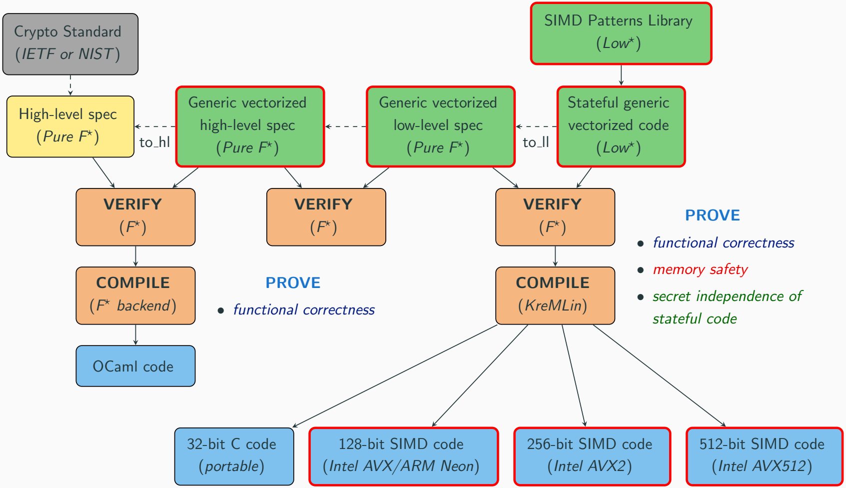
- **High-level spec:** a mathematical spec of a crypto primitive
- **Low-level spec:** a pure spec of an *optimized* algorithm
- **Stateful code:** a Low* impl of the optimized algorithm

HACL×N programming workflow



- **High-level spec:** a mathematical spec of a crypto primitive
- **Generic vectorized high-level spec:**
  a mathematical spec of a *vectorized* algorithm
- **Generic vectorized low-level spec:**
  a pure spec of a *vectorized* algorithm
- **Stateful generic vectorized code:**
  a Low* impl of the vectorized algorithm

HACL×N programming and verification workflow



Parallelizing Polynomial evaluation (Poly1305)

- The main computation in the Poly1305 MAC evaluates the following polynomial over $\mathbb{Z}_p$, where $p = 2^{130} - 5$

$$a = (m_1 \times r^n + m_2 \times r^{n-1} + \ldots + m_n \times r) \bmod p$$

- In practice, Horner's method is used

$$a = (\ldots((0 + m_1) \times r + m_2) \times r + \ldots + m_n) \times r \bmod p$$

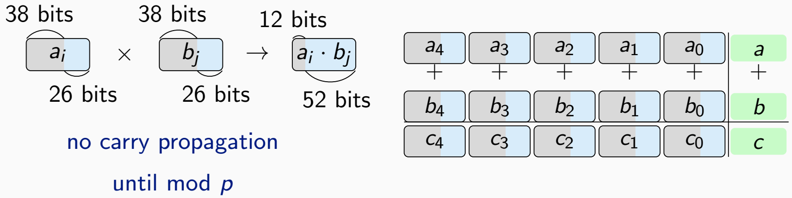
- $w = 2$

$$a_1 = (\ldots((m_1 \times r^2 + m_3) \times r^2 + m_5) \times r^2 + \ldots + m_{n-1}) \mod p$$
$$a_2 = (\ldots((m_2 \times r^2 + m_4) \times r^2 + m_6) \times r^2 + \ldots + m_n) \mod p$$

$$a = (a_1 \times r^2 + a_2 \times r) \mod p$$

Scalar Field Arithmetic for Poly1305

High-level spec	mathematical integers	$c = a + b$
Low-level spec	machine integers	$c_i = a_i + b_i$



- $p = 2^{130} - 5$, each element has up to 130 bits
- The well-known optimization is to use radix-2^{26}

  $a = a_0 + a_1 \cdot 2^{26} + a_2 \cdot 2^{52} + a_3 \cdot 2^{78} + a_4 \cdot 2^{104}$

  a is stored as an array of five **64-bit** unsigned integers
- Modular reduction: $2^{130} \bmod p = 5$
- Implemented in `donna-c32`, `fiat-crypto`, HACL*, etc.

Vectorized Field Arithmetic for Poly1305, $w = 2$

High-level spec	mathematical integers	$c = a+b$ $f = d+e$
Vectorized High-level spec	sequences of mathematical integers	`[c;f] = map2 (+) [a;d] [b;e]`
Vectorized Low-level spec	128-bit machine vector instructions	$[c_i; f_i] = [a_i; d_i] +_v [b_i; e_i]$

$$
\begin{array}{cccccccc}
c & f & & a & d & & b & e \\
\boxed{c_0} & \boxed{f_0} & = & \boxed{a_0} & \boxed{d_0} & +_v & \boxed{b_0} & \boxed{e_0} \\
\boxed{c_1} & \boxed{f_1} & = & \boxed{a_1} & \boxed{d_1} & +_v & \boxed{b_1} & \boxed{e_1} \\
\boxed{c_2} & \boxed{f_2} & = & \boxed{a_2} & \boxed{d_2} & +_v & \boxed{b_2} & \boxed{e_2} \\
\boxed{c_3} & \boxed{f_3} & = & \boxed{a_3} & \boxed{d_3} & +_v & \boxed{b_3} & \boxed{e_3} \\
\boxed{c_4} & \boxed{f_4} & = & \boxed{a_4} & \boxed{d_4} & +_v & \boxed{b_4} & \boxed{e_4}
\end{array}
$$

e.g., $+_v =$`_mm_add_epi64` for Intel AVX

Verified Vectorized Field Arithmetic for Poly1305

```
type field_spec = | M32 | M128 | M256 | M512

let felem (s:field_spec) = lbuffer (sec_uint64xN s) 5ul

val fadd (#s:field_spec) (out f1 f2:felem s) : Stack unit
  (requires λ h →
    live h out ∧ live h f1 ∧ live h f2 ∧
    eq_or_disjoint f1 f2 ∧ eq_or_disjoint f1 out ∧
    eq_or_disjoint f2 out ∧ fadd_pre h f1 f2)
  (ensures λ h0 _ h1 →
    modifies (loc out) h0 h1 ∧ fadd_post h1 out ∧
    feval h1 out == map2 (+%) (feval h0 f1) (feval h0 f2))
```

- Memory Safety
- Functional Correctness
- Secret Independence

Performance Benchmarks

Algorithm	Intel Kaby Lake Laptop			Intel Xeon Workstation			ARM Raspberry Pi 3B+		
	Our Code		Other	Our Code		Other	Our Code		Other
	Scalar	AVX2	Fastest	Scalar	AVX512	Fastest	Scalar	Neon	Fastest
ChaCha20	3.73	0.77	0.75	5.74	0.56	0.56	8.69	5.19	4.49
Poly1305	1.59	0.37	0.35	2.31	0.39	0.51	4.20	3.11	1.50
Blake2b	2.56	2.26	2.02	3.97	3.13	2.84	6.99	–	6.02
Blake2s	4.32	3.34	3.06	6.63	4.52	4.11	11.42	15.30	9.80
$\mathrm{SHA}_{224,256}$	7.41	1.62×8	1.49×8	11.36	1.69×8	2.29×8	15.70	12.92×4	15.09
$\mathrm{SHA}_{384,512}$	5.06	1.95×4	3.25	7.38	1.44×8	4.99	11.27	–	9.77

We measure CPU cycles per byte when processing 16384 bytes.

- Vectorization provides a measurable speedup for all our code on AVX2 and AVX512 ($1.1 - 10\times$)
- Our code is between $3 - 15\%$ slower than the fastest available hand-optimized assembly code on AVX2 and AVX512

Programming and Verification Effort

Algorithm	Coding and Verification Effort (LoC)					Specialized Implementations				
	Scalar Spec	Vec Spec	Equiv Proof	Low* Impl.	Out. C	Portable C code	Arm A64 Neon	Intel x64 AVX	Intel x64 AVX2	Intel x64 AVX512
ChaCha20	151	182	819	510	4083	✓	✓	✓	✓	✓
Poly1305	56 (arith)	122	370 +3594	2361	7136	✓	✓	✓	✓	✓
Blake2b	430	441	324	1077	2824	✓			✓	
Blake2s						✓	✓	✓		
$SHA_{224,256}$	213	420	662	1360	4647	✓	✓	✓	✓	
$SHA_{384,512}$						✓			✓	✓
Total:	850	12242			18690	8	5	5	7	4

- 8 algorithms
- 4 Low* implementations
- 8 portable C implementations
- 21 vectorized implementations for 4 architectures

Contributions

Summary of research contributions:

- the *first* mixed assembly-C verified code
- the *first* verified bignum library suitable for crypto
- the *first* verified implementations of RSA-PSS and FFDHE
- the *first* verified vectorized implementations for ARM Neon and AVX512
- the *first* verified vectorized implementations for Blake2 and SHA-2

Contributions

Summary of research contributions:

- significantly improved speeds for all algorithms in HACL*-v1 (between $3 - 10\times$)
- a more complete HACL*-v2 that now supports high-performance multi-platform implementations of
  - full ciphersuite of TLS 1.3 (Chacha20-Poly1305, X25519, SHA-2, RSA-PSS)
  - other protocols like WireGuard

Deployment

Performance Improvements via Formally-Verified Cryptography in Firefox

Kevin Jacobs and Benjamin Beurdouche | July 6, 2020

Improving the implementation of cryptography in Tezos Octez

in-depth | 14 October 2021 | Nomadic Labs

- Various algorithms from our verified cryptographic library are already deployed in
  - Mozilla's NSS cryptographic library
  - Tezos blockchain
  - Wireguard VPN
  - Zinc crypto library for the Linux Kernel, etc.
- All our code is publicly available at

  `https://github.com/project-everest/hacl-star`

Limitations and Future work

- Programming and Verification effort
- Protections against side-channel attacks
- The coverage of algorithms
  - Post-Quantum Cryptography
  - Lightweight Cryptography
  - Zero-Knowledge Proofs, etc.

Verified Implementation of Post-Quantum Cryptography

The need for constant-time and highly-optimized functionally-correct code for newly designed constructions is dire

- e.g., an exploitable timing leakage was found in the official reference implementation of FrodoKEM

```
// If (Bp == BBp & C == CC) then ss = F(ct || k'), else ss = F(ct || s)
// Needs to avoid branching on secret data as per:
//     Qian Guo, Thomas Johansson, Alexander Nilsson. A key-recovery timing attack on post-quantum
//     primitives using the Fujisaki-Okamoto transformation and its application on FrodoKEM. In CRYPTO 2020.
int8_t selector = ct_verify(Bp, BBp, PARAMS_N*PARAMS_NBAR) | ct_verify(C, CC, PARAMS_NBAR*PARAMS_NBAR);
// If (selector == 0) then load k' to do ss = F(ct || k'), else if (selector == -1) load s to do ss = F(ct || s)
ct_select((uint8_t*)Fin_k, (uint8_t*)kprime, (uint8_t*)sk_s, CRYPTO_BYTES, selector);
```

- As a first case study, we have built formally verified portable C implementations for all versions of FrodoKEM