Part 3:
Towards High-Assurance Cryptographic Software
Modern applications employ crypto to protect sensitive data and transactions

- **Protocols**: TLS, SSH, Signal, OpenID
- **End-to-End encryption**: cloud storage, ...

Coding with crypto is easy to get wrong, even for security experts!

Formal verification can provide higher assurance (and find new attacks)
What goes wrong in TLS?

Legacy, obsolete, broken crypto
• RC4, RSA PKCS#1v1.5, 3DES, MD5

Protocol design flaws
• 3Shake, Logjam, SLOTH

Implementation bugs
• HeartBleed, GotoFail, SKIP, FREAK
Remember these?

Unexpected state transitions in OpenSSL, NSS, Java, SecureTransport, ...

- Required messages are allowed to be skipped
- Unexpected messages are allowed to be received
- CVEs for many libraries

How come all these bugs?

- In independent code bases, sitting in there for years
- Are they exploitable?
Common Implementation Bugs

Bugs in Crypto Primitives
• Incorrect results, side-channels

Bugs in Crypto Usage
• Nonce reuse, Mac-Encode-Encrypt

Bugs in Protocol State Machine
• Incorrect composition

Bugs in Packet Parsing
• Buffer overrun
How do we fix it?

Verify protocol software
- Verified Crypto Library (HACL*)
- Verified TLS library (miTLS)

Today:
- Brief intro to HACL*
- Briefer intro to miTLS
Reading Materials and Tools

- **F* language and tutorial**: [http://fstar-lang.org](http://fstar-lang.org)

- **Proving the TLS Handshake Secure (As It Is)**. CRYPTO 2014.

- **Dependent Types and Multi-Monadic Effects in F***. ACM POPL 2016.


- **Implementing and Proving the TLS 1.3 Record Layer**. IEEE S&P 2017.
Verified Crypto Libraries:
HACL*
Modular Arithmetic in Crypto

\[ g^{xy} \mod p \]

- \(0 < x, y < p\)
- \(g\) fixed
- Large Prime
  - (e.g. \(2^{255} - 19\))
Implementing Modular Exponentiation

\[ a^b \mod n \]

\[ = a \times a \times \ldots \times a \mod n \]

Big Integer (≥ 256-bit) arithmetic
Textbook Multiplication

\[
\begin{array}{c}
1101 \\
\times \\
1010 \\
\hline \\
0000 \\
1101 \\
0000 \\
\hline \\
1101 \\
\hline \\
10000010
\end{array}
\]
256-bit Modular Multiplication on 64-bit Computers
256-bit Modular Multiplication on 64-bit Computers

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

How expensive is it?
• Dominates crypto overhead
• \(n^2\) 64x64 multiplications
• Long intermediate arrays
• Many carry steps
Optimizations vs. Side-Channel Leaks

Skipping steps is faster
• Fewer additions, carries

... but may leak information
• Runtime proportional to number of 1s in 1010
• Attacker can observe runtime to guess input
• Input may be secret key!
Many prime-specific optimizations
• Trade-off multiplication vs modular reduction
• Use only 51 out of 64 bits to avoid carries
• Precompute reusable intermediate values
• Parallelize (vectorize) multiplication and squaring

Complex optimizations increase chances of bugs!
Unsaturated Arithmetic for Curve25519

Carries propagated as infrequently as possible

Add without carry

Reduce modulo $2^{255} - 19$
Many bugs in optimized bignum code

[2013] Bug in amd-64-64-24k Curve25519
   “Partial audits have revealed a bug in this software \((r1 += 0 + \text{carry should be } r2 += 0 + \text{carry in amd64-64-24k})\) that would not be caught by random tests; this illustrates the importance of audits.”

[2014] Arithmetic bug in TweetNaCl’s Curve25519

[2014] Carry bug in Langley’s Donna-32 Curve25519

[2016] Arithmetic bug in OpenSSL Poly1305

[2017] Arithmetic bug in Mozilla NSS GF128

+ Many memory bugs, side-channel leaks, ...
Towards High-Assurance Crypto Software

Crypto code is easy to get wrong and hard to test well

- side-channel leaks [CVE-2018-5407, CVE-2018-0737]
- arithmetic bugs [CVE-2017-3732, CVE-2017-3736]

Formal verification can systematically prevent bugs

- Many tools: F*, Cryptol/Saw, VST, Fiat-Crypto, Vale, Jasmin
- But verification often requires (PhD-level) manual effort

How do we scale verification up to full crypto libraries?

- Low-level platform specific optimizations for a suite of algorithms
Writing Verified Crypto Code

CRYPTO STANDARD (IETF/NIST) → ALGORITHM PSEUDOCODE → IMPLEMENTATION (C, 200 loc)

```c
static void chacha20_core(chacha_buf *output, const u32 input[16])
{
    u32 x[16];
    int i;
    const union {
        long one;
        char little;
    } is_endian = { 1 };

    memcpy(x, input, sizeof(x));

    for (i = 20; i > 0; i -= 2) {
        QUARTERROUND(0, 4, 8, 12);
        QUARTERROUND(1, 5, 9, 13);
        QUARTERROUND(2, 6, 10, 14);
        QUARTERROUND(3, 7, 11, 15);
        QUARTERROUND(0, 5, 10, 15);
        QUARTERROUND(1, 6, 11, 12);
        QUARTERROUND(2, 7, 8, 13);
        QUARTERROUND(3, 4, 9, 14);
    }
}
```

Obviously correct? unless we introduced a buffer overflow, or a timing leak
Writing Verified Crypto Code

CRYPTO STANDARD (IETF/NIST)

ALGORITHM PSEUDOCODE

IMPLEMENTATION (C, 500 loc)

while (len >= POLY1305_BLOCK_SIZE) {
    /* h += m[i] */
    h0 = (u32)(d0 = (u64)h0 + U8TOU32(inp + 0));
    h1 = (u32)(d1 = (u64)h1 + (d0 >> 32) + U8TOU32(inp + 4));
    h2 = (u32)(d2 = (u64)h2 + (d1 >> 32) + U8TOU32(inp + 8));
    h3 = (u32)(d3 = (u64)h3 + (d2 >> 32) + U8TOU32(inp + 12));
    h4 += (u32)(d3 >> 32) + padbit;

    /* h += r t% p, where "%" stands for "partial remainder" */
    d0 = ((u64)h0 * r0) +
        ((u64)h1 * s3) +
        ((u64)h2 * s2) +
        ((u64)h3 * s1);
    d1 = ((u64)h0 * r1) +
        ((u64)h1 * r0) +
        ((u64)h2 * s3) +
        ((u64)h3 * s2) +
        (h4 * s1);
    d2 = ((u64)h0 * r2) +
        ((u64)h1 * r1) +
        ((u64)h2 * r0) +
        ((u64)h3 * s3) +
        (h4 * s2);
    d3 = ((u64)h0 * r3) +
        ((u64)h1 * r2) +
        ((u64)h2 * r1) +
        ((u64)h3 * r0) +
        (h4 * s3);
    h4 = (h4 * r0);
Writing Verified Crypto Code

CRYPTO STANDARD (IETF/NIST)

ALGORITHM PSEUDOCODE

FORMAL SPEC (F*/CRYPTOL/Coq)

IMPLEMENTATION (C, 500 loc)

Verification Guarantees
1. Functional Correctness
2. Memory Safety
3. Secret Independence (constant-time)
HACL*: a verified C crypto library

[Zinzindohoe et al. ACM CCS 2017]

A growing library of verified crypto algorithms

- Curve25519, Ed25519, Chacha20, Poly1305, SHA-2, HMAC, ...

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 spec written in F*

Generates readable, portable, standalone C code

- Performance comparable to hand-written C crypto libraries
- Used in Mozilla Firefox, WireGuard VPN, Tezos Blockchain, ...

[https://github.com/project-everest/hacl-star](https://github.com/project-everest/hacl-star)
Potential memory safety bug, or functional correctness bug, or side-channel leak.

Source code not in Low*; Cannot be compiled to C.
F*: a verification oriented language

F* features:
- Functional language (« à la Ocaml »)
- Customizable verification system (« à la Coq »)
- Proof automation via SMT solvers (Z3)
- Compilation tools to Ocaml, F#, C

http://fstar-lang.org
Basis for Verification: **Refinement types**

A *refinement type* is a base type qualified with a logical formula; the formula can express invariants, preconditions, postconditions, …

Refinement types are types of the form $x : T \{ C \}$ where
- $T$ is the base type,
- $x$ refers to the result of the expression, and
- $C$ is a logical formula

The values of this type are the values $M$ of type $T$ such that $C{M/x}$ holds.
// Sample type and value declarations in F*

```fsharp
type nat = n:int{ 0 <= n }
val read: n:nat -> b:bytes{ Length(b) <= n }
```

// Sample cryptographic library interface in F*

```fsharp
module AES

// abstract type for secrets

type key

// abstract type for secrets

type block = b:bytes{ Length(b)=16 }

val encrypt: k:key -> p:block -> c:block {c=AES(k,p)}
val decrypt: k:key -> c:block -> p:block {c=AES(k,p)}
```
Modular Typing & Runtime Safety

Safety means that all logical refinements hold at runtime.

**THEOREM** (Safety). If \( \emptyset \vdash A : T \) then \( A \) is safe.

We write \( I_0 \vdash A \sim I \) when, in the typing environment \( I_0 \), the module \( A \) is well-typed and exports the interface \( I \).

If \( \emptyset \vdash A_0 \sim I_0 \) and \( I_0 \vdash A : T \), then \( \emptyset \vdash A_0 \cdot A : T \).
F*: verification by typing

- We program and verify in F*
- F* typechecks code and calls Z3, an SMT solver, on each logical proof obligation
Example: Poly1305 MAC Algorithm

Poly1305 is a message authentication code

\[ poly(k, m, w_1...w_n) = m + w_1 k^1 + ... + w_n k^n \mod (2^{130} - 5) \]

It authenticates a data stream \( w_1...w_n \) by

- Encoding it as a polynomial in the prime-field modulo \( 2^{130} - 5 \)
- Evaluating it at a point \( k \) (first part of the key)
- Masking the result with \( m \) (second part of the key)
Specifying Poly1305 in Pure F*

- Short, easy to review
- Uses arbitrary precision natural numbers
- Compiles to OCaml
- Passes RFC test-vectors
Implementing Poly1305 in Stateful F*

1. Encode field elements with a 44-44-42 unsaturated representation
2. Factor out generic bignum operations (+, *) into a shared library
3. Optimized prime-specific field arithmetic in Poly1305
4. Expose an Init-Update-Finish API for incremental use

Low* code+proofs: 1508 lines (generic bignum) + 3208 lines (poly1305)
Compiled C code: 451 lines
Verifying Memory Safety by Typing

1. Ensure all pointers are live (initialized and not yet freed)
2. Verify all array accesses (access within bounds)
3. Enforce disjointness (needed for correctness)
4. Track modifications (needed for composability)
Verifying Functional Correctness

Prove that stateful code matches pure F* specification

• Relies on mathematical theory of modular arithmetic
• Simple arithmetic goals automatically verified by the SMT solver (Z3)
• Complex prime-specific optimizations proved using F*
Verifying Secret Independence

Type-based “constant-time” coding discipline

• Code cannot branch on secrets
• Code cannot use secret indices to lookup arrays
• Essentially, a crude static information flow checker via types

Prevents timing attacks within C semantics

• No guarantees on compiled assembly
• Does not guarantee absence of other side channels
• For better guarantees, see:

  Verifying Constant-Time Implementations, Almeida et al. Usenix’16
Verifying Secret Independence

- Abstract types for opaque “hidden integers”
  their concrete values are only available in specifications
- Allowed constant-time operations: +, -, *, ^, &, |
  but no comparisons or divisions, or use as array indexes
  (Forbidden operations depend on target platform)

```plaintext
abstract type hint64_t
val v: hint64_t \to\ GTot\ uint64_t

val (+): hint64_t \to\ hint64_t \to\ Tot\ hint64_t
val (^): hint64_t \to\ hint64_t \to\ Tot\ hint64_t
// val (/): hint64_t \to\ hint64_t \to\ Tot\ hint64_t

type key = b:buffer\ uint64_t\{\text{length}\ \ b = 32\}
```
HACL* Verification Workflow

Standard (RFC, NIST) ➔ Spec (Pure F*) ➔ Compile (KreMLin) ➔ Verify (F*) ➔ Code (Stateful F*) ➔ Optimized Code (C)

- Potential memory safety bug, or functional correctness bug, or side-channel leak.
- Source code not in Low*; Cannot be compiled to C.
Implementing Poly1305

Compiled C Code

F* Code for Poly1305

Compiled C Code

static void Hacl_Impl_Poly1305_64_poly1305_last_pass(uint64_t *acc)

Hacl_Bignum_Fproduct_carry_limb(acc);
Hacl_Bignum_Modulo_carry_top(acc);

uint64_t a0 = acc[0];
uint64_t a10 = acc[1];
uint64_t a20 = acc[2];

uint64_t a0 = a0 & (uint64_t)0xffffffff;
uint64_t a10 = a10 >> (uint32_t)144;
uint64_t a20 = a20 & (uint64_t)0xffffffff;

uint64_t r1 = (a10 + r1) >> (uint32_t)144;
acc[0] = a0;
acc[1] = a1;
acc[2] = a2;

Hacl_Bignum_Modulo_carry_top(acc);

uint64_t t0 = acc[0];
uint64_t t1 = acc[1];
uint64_t t2 = acc[2];

uint64_t t0 = (uint64_t)0 & (uint64_t)144 - (uint64_t)1);
uint64_t t1 = (t1 & (uint64_t)0) >> (uint32_t)144);

acc[0] = 0;
acc[1] = 1;
acc[2] = 2;

uint64_t t0 = acc[0];
uint64_t t1 = acc[1];
uint64_t t2 = acc[2];

uint64_t t0 = (uint64_t)0 & (uint64_t)144 - (uint64_t)1);
uint64_t t1 = (t1 & (uint64_t)0) >> (uint32_t)144);

acc[0] = 0;
acc[1] = 1;
acc[2] = 2;
**Low**: High-level verification for low-level code

[Protzenko et al. ICFP 2017]

Low* is a subset of F* that mimics C programs
- Relies on a C-like memory model;
- Mostly *first-order code* with *combinators* to get C loops;
- A few *low-level libraries* for arrays, structs, and C base types.

When writing proofs and specifications, the programmer
- uses all of F*, including higher-order functions and polymorphism;
- proves *memory safety, correctness, cryptographic security*, by adding lemmas, type annotations, auxiliary functions, etc.;
- compiles the code to a first-order program by *erasing all proofs*.

**Motto**: the code is low-level but the verification is not
KreMLin: Compiling Low* to Readable C

https://github.com/FStarLang/kremlin

- Implements a formal translation from Low* to Clight
- Hand-written proof of trace preservation
  - Compilation preserves side-channel guarantees [Protzenko et al. ICFP 2017]
- Lots of engineering to generate readable, efficient C code
Implementing Poly1305 in Low* Code for Poly1305

Compiled C Code

Low* Code for Poly1305

Compiled C Code

HACL*
**HACL* Verification**

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Spec (F* loc)</th>
<th>Code+Proofs (Low* loc)</th>
<th>C Code (C loc)</th>
<th>Verification (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Salsa20</td>
<td>70</td>
<td>651</td>
<td>372</td>
<td>280</td>
</tr>
<tr>
<td>Chacha20</td>
<td>70</td>
<td>691</td>
<td>243</td>
<td>336</td>
</tr>
<tr>
<td>Chacha20-Vec</td>
<td>100</td>
<td>1656</td>
<td>355</td>
<td>614</td>
</tr>
<tr>
<td>SHA-256</td>
<td>96</td>
<td>622</td>
<td>313</td>
<td>798</td>
</tr>
<tr>
<td>SHA-512</td>
<td>120</td>
<td>737</td>
<td>357</td>
<td>1565</td>
</tr>
<tr>
<td>HMAC</td>
<td>38</td>
<td>215</td>
<td>28</td>
<td>512</td>
</tr>
<tr>
<td>Bignum-lib</td>
<td>-</td>
<td>1508</td>
<td>-</td>
<td>264</td>
</tr>
<tr>
<td>Poly1305</td>
<td>45</td>
<td>3208</td>
<td>451</td>
<td>915</td>
</tr>
<tr>
<td>X25519-lib</td>
<td>-</td>
<td>3849</td>
<td>-</td>
<td>768</td>
</tr>
<tr>
<td>Curve25519</td>
<td>73</td>
<td>1901</td>
<td>798</td>
<td>246</td>
</tr>
<tr>
<td>Ed25519</td>
<td>148</td>
<td>7219</td>
<td>2479</td>
<td>2118</td>
</tr>
<tr>
<td>AEAD</td>
<td>41</td>
<td>309</td>
<td>100</td>
<td>606</td>
</tr>
<tr>
<td>SecretBox</td>
<td>-</td>
<td>171</td>
<td>132</td>
<td>62</td>
</tr>
<tr>
<td>Box</td>
<td>-</td>
<td>188</td>
<td>270</td>
<td>43</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>801</strong></td>
<td><strong>22,926</strong></td>
<td><strong>7,225</strong></td>
<td><strong>9127</strong></td>
</tr>
</tbody>
</table>

Table 1: HACL* code size and verification times

- Verified vectorized crypto primitive
- Sharing verification effort across Poly1305, Ed25519, Curve25519
- Unreasonably large effort for simple goals
HACL* Verification

• Share verified libraries across various primitives
• Verified optimizations SIMD vectorization, prime-specific arithmetic
• Significant manual effort in initial release, now significantly reduced.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Spec (F* loc)</th>
<th>Code+Proofs (Low* loc)</th>
<th>C Code (C loc)</th>
<th>Verification (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Salsa20</td>
<td>70</td>
<td>651</td>
<td>372</td>
<td>280</td>
</tr>
<tr>
<td>Chacha20</td>
<td>70</td>
<td>691</td>
<td>243</td>
<td>336</td>
</tr>
<tr>
<td>Chacha20-Vc</td>
<td>100</td>
<td>1656</td>
<td>355</td>
<td>614</td>
</tr>
<tr>
<td>SHA-256</td>
<td>96</td>
<td>622</td>
<td>313</td>
<td>798</td>
</tr>
<tr>
<td>SHA-512</td>
<td>120</td>
<td>737</td>
<td>357</td>
<td>1565</td>
</tr>
<tr>
<td>HMAC</td>
<td>38</td>
<td>215</td>
<td>28</td>
<td>512</td>
</tr>
<tr>
<td>Bignum-lib</td>
<td>-</td>
<td>1508</td>
<td>-</td>
<td>264</td>
</tr>
<tr>
<td>Poly1305</td>
<td>45</td>
<td>3208</td>
<td>451</td>
<td>915</td>
</tr>
<tr>
<td>X25519-lib</td>
<td>-</td>
<td>3849</td>
<td>-</td>
<td>768</td>
</tr>
<tr>
<td>Curve25519</td>
<td>73</td>
<td>1901</td>
<td>798</td>
<td>246</td>
</tr>
<tr>
<td>Ed25519</td>
<td>148</td>
<td>7219</td>
<td>2479</td>
<td>2118</td>
</tr>
<tr>
<td>AEAD</td>
<td>41</td>
<td>309</td>
<td>100</td>
<td>606</td>
</tr>
<tr>
<td>SecretBox</td>
<td>-</td>
<td>171</td>
<td>132</td>
<td>62</td>
</tr>
<tr>
<td>Box</td>
<td>-</td>
<td>188</td>
<td>270</td>
<td>43</td>
</tr>
<tr>
<td>Total</td>
<td>801</td>
<td>22,926</td>
<td>7,225</td>
<td>9127</td>
</tr>
</tbody>
</table>

Table 1: HACL* code size and verification times
HACL* Performance

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>HACL*</th>
<th>OpenSSL</th>
<th>libsodium</th>
<th>TweetNaCl</th>
<th>OpenSSL (asm)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHA-256</td>
<td>13.43</td>
<td>16.11</td>
<td>12.00</td>
<td>-</td>
<td>7.77</td>
</tr>
<tr>
<td>SHA-512</td>
<td>8.09</td>
<td>10.34</td>
<td>8.06</td>
<td>12.46</td>
<td>5.28</td>
</tr>
<tr>
<td>Salsa20</td>
<td>6.26</td>
<td>-</td>
<td>8.41</td>
<td>15.28</td>
<td>-</td>
</tr>
<tr>
<td>ChaCha20</td>
<td>6.37 (ref)</td>
<td>7.84</td>
<td>6.96</td>
<td>-</td>
<td>1.24</td>
</tr>
<tr>
<td>Poly1305</td>
<td>2.19</td>
<td>2.16</td>
<td>2.48</td>
<td>32.65</td>
<td>0.67</td>
</tr>
<tr>
<td>Curve25519</td>
<td>154,580</td>
<td>358,764</td>
<td>162,184</td>
<td>2,108,716</td>
<td>-</td>
</tr>
<tr>
<td>Ed25519 sign</td>
<td>63.80</td>
<td>-</td>
<td>24.88</td>
<td>286.25</td>
<td>-</td>
</tr>
<tr>
<td>Ed25519 verify</td>
<td>57.42</td>
<td>-</td>
<td>32.27</td>
<td>536.27</td>
<td>-</td>
</tr>
<tr>
<td>AEAD</td>
<td>8.56 (ref)</td>
<td>8.55</td>
<td>9.60</td>
<td>-</td>
<td>2.00</td>
</tr>
<tr>
<td>SecretBox</td>
<td>8.23</td>
<td>-</td>
<td>11.03</td>
<td>47.75</td>
<td>-</td>
</tr>
<tr>
<td>Box</td>
<td>21.24</td>
<td>-</td>
<td>21.04</td>
<td>148.79</td>
<td>-</td>
</tr>
</tbody>
</table>

- 20% faster than previous code in Firefox
- As fast as hand-optimized C code in OpenSSL
- 0%-30% slower than equivalent hand-tuned assembly
Can we make it faster?
Verified Assembly, Vectorized Algorithms

Can we make it easier?
Smaller Specs and Code, Fewer Proof Annotations
## HACL*: estimating verification effort

<table>
<thead>
<tr>
<th></th>
<th>CHACHA20</th>
<th>POLY1305</th>
</tr>
</thead>
<tbody>
<tr>
<td>High-level F* Spec</td>
<td>70 lines</td>
<td>45 lines</td>
</tr>
<tr>
<td>Verified F* Code</td>
<td>691 lines</td>
<td>3967 lines</td>
</tr>
<tr>
<td>Generated C Code</td>
<td>285 lines</td>
<td>451 lines</td>
</tr>
<tr>
<td>Proof Annotations</td>
<td>406 lines</td>
<td>3516 lines</td>
</tr>
</tbody>
</table>

Every line of verified C requires 2x-7x lines of proof

Complex mathematical reasoning interleaved with many boring steps
Platform-Specific Optimizations

**CRYPTO STANDARD** (IETF/NIST)

**ALGORITHM PSEUDOCODE**

**HIGH-LEVEL SPEC** (F*/CRYPTOL/Coq)

**PORTABLE 32-BIT** (C, 500 loc)

**64-BIT** (C, 200 loc)

**INTEL AVX** (ASM, 1 Kloc)

**INTEL AVX2** (ASM, 1 Kloc)

**ARM NEON** (ASM, 1 Kloc)

---

**Idea 1:** Verify optimized assembly

- Write perf-critical bits in assembly
- Write higher-level bits in C
- Verify and compose the two

A hard target for formal verification (probably also awful for maintenance)
Recall: Unsaturated Arithmetic for Curve25519

9 more $64 \times 64 \rightarrow 128$ multiplications
but still faster because of less carry propagation

add without carry
reduce modulo $2^{255} - 19$
Saturated 64-bit Arithmetic with Intel ADX
[Oliveira et al, SAC 2017]

An instruction set with 2 carry flags can significantly reduce the cost of carry propagation!

New speed record?
Measurements using Jason Donenfeld’s Linux Kernel Benchmarking Suite for WireGuard.

Intel(R) Xeon(R) CPU E3-1505M v5 @ 2.80GHz
donna64: 160942 cycles per call
haci64: 140902 cycles per call
fiat64: 144106 cycles per call
sandy2x: 136074 cycles per call
precomp_bmi2: 121350 cycles per call
precomp_adx: 117676 cycles per call
amid64: 143628 cycles per call
fiat32: 307971 cycles per call
donna32: 544254 cycles per call
Saturated 64-bit Arithmetic with Intel ADX

Armando Faz Hernández

Post by Jason A. Donenfeld
Hi Armando,
I've started importing your precomputation implementation into kernel space for use in kbench9000 (and in WireGuard and the kernel crypto library too, of course).
- The first problem remains the license. The kernel requires GPLv2-compatible code. GPLv3 isn't compatible with GPLv2. This isn't up to me at all, unfortunately, so this stuff will have to be licensed differently in order to be useful.

The rfc7748_precomputed library is now released under LGPLv2.1.
We are happy to see our code integrated in more projects.

Post by Jason A. Donenfeld
- It looks like the precomputation implementation is failing some unit tests! Perhaps it's not properly reducing incoming public points? There's the vector if you'd like to play with it. The other test vectors I have do pass, though, which is good I suppose.

Thanks, for this observation. The code was missing to handle some carry bits, producing incorrect outputs for numbers between 2p and 2^256. Now, I have rewritten some operations for GF(2^255-19) considering all of these cases. More tests were added and fuzz test against HACL implementation.
Vale: extensible, assembly language verification
[Usenix 2017, POPL 2019]

machine model (F*)

instructions
- type reg = r0 | r1

semantics
- eval(Mov(dst, src), ...) = ...
- eval(Add(dst, src), ...) = ...
- eval(Neg(dst), ...) = ...

code generation
- print(Mov(dst, src), ...) = "mov " + (...dst) + (...src)
- print(Add(dst, src), ...) = ...

Trust Computing Base

crypto spec
- mem[eax] == SHA(mem[ebx])

Vale

code
- [Mov(r1, r0), Add(r1, r0), Add(r1, r1)]
- lemma_mov(...);
- lemma_add(…);

proof
- lemma_add(…);

Vale Crypto code

machine interface
- procedure mov(...)
  - requires ...
  - ensures ...
  - {
    ...
  }
- procedure add(...)

program
- procedure quadruple(…)
  - requires 0 <= r0 < 2^{30};
  - ensures r1 == r0 * 4;
  - {
    mov(r1, r0);
    add(r1, r0);
    add(r1, r1);
  }
Verified Assembly for 256-bit Multiplication

Implement core arithmetic in Vale x86
- Use ADX+BMI2 instructions
- Verify correctness + constant-time

Implement curve operations in HACL*
- Use standard C coding style
- Optimize add/double formulas, montgomery ladder, etc.

Prove that composition of Vale and F* code meets Curve25519 spec
EverCrypt: a crypto provider with Vale + HACL*

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>C version</th>
<th>Targeted ASM version</th>
</tr>
</thead>
<tbody>
<tr>
<td>AEAD</td>
<td></td>
<td>AES-NI + PCLMULQDQ + AVX</td>
</tr>
<tr>
<td>AES-GCM</td>
<td>yes</td>
<td>AES-NI + PCLMULQDQ + AVX</td>
</tr>
<tr>
<td>ChachaPoly</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hashes</td>
<td>yes</td>
<td>SHA-EXT (for SHA2-224+SHA2-256)</td>
</tr>
<tr>
<td>MD5</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>SHA1</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>SHA2</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>MACS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HMAC</td>
<td>yes</td>
<td>agile over hash</td>
</tr>
<tr>
<td>Poly1305</td>
<td>yes</td>
<td>X64</td>
</tr>
<tr>
<td>Key Derivation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HKDF</td>
<td>yes</td>
<td>agile over hash</td>
</tr>
<tr>
<td>ECC</td>
<td>yes</td>
<td>BMI2 + ADX</td>
</tr>
<tr>
<td>Curve25519</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ciphers</td>
<td>yes</td>
<td>AES NI + AVX</td>
</tr>
<tr>
<td>Chacha20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AES128, 256</td>
<td></td>
<td>AES NI + AVX</td>
</tr>
<tr>
<td>AES-CTR</td>
<td></td>
<td>AES NI + AVX</td>
</tr>
</tbody>
</table>

C Compiler
## Curve25519 Performance: Vale + HACL*

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Radix</th>
<th>Language</th>
<th>CPU cy.</th>
</tr>
</thead>
<tbody>
<tr>
<td>donna64</td>
<td>51</td>
<td>64-bit C</td>
<td>159634</td>
</tr>
<tr>
<td>fiat-crypto [24]</td>
<td>51</td>
<td>64-bit C</td>
<td>145248</td>
</tr>
<tr>
<td>amd64-64</td>
<td>51</td>
<td>Intel x86_64 asm</td>
<td>143302</td>
</tr>
<tr>
<td>sandy2x</td>
<td>25.5</td>
<td>Intel AVX asm</td>
<td>135660</td>
</tr>
<tr>
<td><em><em>HACL</em> portable</em>*</td>
<td>51</td>
<td>64-bit C</td>
<td>135636</td>
</tr>
<tr>
<td>openssl*</td>
<td>64</td>
<td>Intel ADX asm</td>
<td>118604</td>
</tr>
<tr>
<td>Oliveira et al. [45]</td>
<td>64</td>
<td>Intel ADX asm</td>
<td>115122</td>
</tr>
<tr>
<td><strong>EverCrypt: Vale + HACL</strong>*</td>
<td>64</td>
<td>64-bit C + Intel ADX asm</td>
<td>113614</td>
</tr>
</tbody>
</table>

Figure 11. Performance comparison between Curve25519 Implementations.
Other Platform-Specific Optimizations

**CRYPTO STANDARD (IETF/NIST)**

**ALGORITHM PSEUDOCODE**

**HIGH-LEVEL SPEC (F*/CRYPTOL/Coq)**

**PORTABLE 32-BIT (C, 500 loc)**

**64-BIT (C, 200 loc)**

**INTEL AVX (ASM, 1 Kloc)**

**INTEL AVX2 (ASM, 1 Kloc)**

**ARM NEON (ASM, 1 Kloc)**

*Ideas:*

1. **Large code base for each algorithm**
   - Independently evolving implementations
   - Different algorithmic optimizations
   - Hard to tell which version will be used
   - A hard target for formal verification (probably also awful for maintenance)

2. **Write & verify generic SIMD code**
   - Compiles to platform-specific code
   - A **single target** for formal verification
   - A basepoint for further optimization
## HACL* Vectorization Performance

### CHACHA20

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>32-bit Scalar</td>
<td>4 cy/b</td>
<td></td>
</tr>
<tr>
<td>128-bit Vectorized (AVX)</td>
<td>1.5 cy/b</td>
<td></td>
</tr>
<tr>
<td>256-bit Vectorized (AVX2)</td>
<td><strong>0.79 cy/b</strong></td>
<td></td>
</tr>
<tr>
<td>Fastest Assembly (OpenSSL AVX2)</td>
<td><strong>0.75 cy/b</strong></td>
<td></td>
</tr>
</tbody>
</table>

### POLY1305

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>32-bit Scalar</td>
<td>1.5 cy/b</td>
<td></td>
</tr>
<tr>
<td>128-bit Vectorized (AVX)</td>
<td>0.75 cy/b</td>
<td></td>
</tr>
<tr>
<td>256-bit Vectorized (AVX2)</td>
<td><strong>0.39 cy/b</strong></td>
<td></td>
</tr>
<tr>
<td>Fastest Assembly (OpenSSL AVX2)</td>
<td><strong>0.34 cy/b</strong></td>
<td></td>
</tr>
</tbody>
</table>

Measurements with gcc-7 on Intel i7-7560 (Skylake) running Ubuntu 18.10
## Estimating Verification Effort

### CHACHA20

<table>
<thead>
<tr>
<th>Component</th>
<th>Effort</th>
</tr>
</thead>
<tbody>
<tr>
<td>hacsapc</td>
<td>150 lines</td>
</tr>
<tr>
<td>Vectorized algorithm</td>
<td>500 lines</td>
</tr>
<tr>
<td>Correctness proofs</td>
<td>700 lines</td>
</tr>
<tr>
<td>Vectorized code</td>
<td>500 lines</td>
</tr>
<tr>
<td><strong>Total Proof Effort</strong></td>
<td>1700 lines</td>
</tr>
<tr>
<td><strong>Generated C code</strong></td>
<td>3700 lines</td>
</tr>
</tbody>
</table>

### POLY1305

<table>
<thead>
<tr>
<th>Component</th>
<th>Effort</th>
</tr>
</thead>
<tbody>
<tr>
<td>hacsapc</td>
<td>80 lines</td>
</tr>
<tr>
<td>Vectorized algorithm</td>
<td>450 lines</td>
</tr>
<tr>
<td>Correctness proofs</td>
<td>2000 lines</td>
</tr>
<tr>
<td>Vectorized code</td>
<td>1500 lines</td>
</tr>
<tr>
<td><strong>Total Proof Effort</strong></td>
<td>4000 lines</td>
</tr>
<tr>
<td><strong>Generated C code</strong></td>
<td>16000 lines</td>
</tr>
</tbody>
</table>

Effort roughly the same as verifying one implementation.
HACL* in Mozilla Firefox 57
HACL* in Project Everest

Target: verified HTTPS Stack

- HACL* as a verified crypto provider
- Integrates in F* with **protocol proofs** and **cryptographic proofs**
  - Implementing and Proving the TLS 1.3 Record Layer, Bhargavan et al, Oakland 2017
- Interops with **Vale assembly** at the F* specification level
  - Vale: Verifying High-Performance Cryptographic Assembly Code, Bond et al, USENIX17
Verified Crypto Protocol Code:
miTLS in F*
miTLS [2012-2018]

A verified reference implementation of TLS

- Covers TLS 1.0-1.2 (TLS 1.3 ongoing)
- Covers all major protocol modes and ciphersuites
miTLS Security theorem

Main crypto result: concrete TLS & ideal TLS are computationally indistinguishable

We prove that ideal miTLS meets its secure channel specification using standard program verification (typing)
Modular Type-Based Cryptographic Verification

- **MAC (SHA1)**
- **symmetric encryption (AES-CBC)**
- **symmetric encryption (RC4)**

**Encrypt then-MAC**

**fragment-MAC-encode-then-encrypt**

**secure channel**

- **secure RPC**
- **TLS 1.2**

- **some attack**
- **another attack**

- **cryptographic algorithms**
- **typed interfaces:**
  - **cryptographic assumptions**
- **cryptographic constructions**
- **typed interfaces:**
  - **security guarantees**
- **security protocols**
- **typed interfaces:**
  - **attacker models**
- **active adversaries**
Sample modular verification (protocol)

**RPC protocol using Authenticated Encryption**

- **Adversary Model**: any typed F# program
- **Active adversaries**: any typed F# program
- **Secure RPC**: any typed F# program
- **RPC API**: any typed F# program
- **Formatting**: message format
- **Bytes**: system libraries
- **Networking**: security protocols
Sample modular verification (crypto)

**RPC using Encrypt-then-MAC**

- **Cryptographic schemes**
- **Cryptographic constructions**
- **Probabilistic computational indistinguishability**

**Encrypt-then-MAC**
- **Authenticated encryption**
- **Indistinguishability under chosen-plaintext attacks (IND-CPA)**
- **Indistinguishability under chosen-message attacks (INT-CMA)**
- **Equivalent to ideal schemes**

**Secure RPC**
- **RPC API**
- **System libraries**
- **Networking**

**Adversary Model**
- **Active F# program**
- **Any typed F# program**

**Any typed application code**

- **Active adversaries**
Sample functionality:
Message Authentication Codes

module MAC

val macsize : bytes

val GEN : unit -> key

val MAC : key -> text -> mac

val VERIFY : key -> text -> mac -> bool

This interface says nothing on the security of MACs.
MAC keys are abstract

Sample functionality:
Message Authentication Codes

module MAC

  type text = bytes  val macsize
  type key
  type mac  = bytes

  val GEN : unit -> key
  val MAC : key -> text -> mac
  val VERIFY: key -> text -> mac -> bool
MAC keys are abstract

Sample functionality:
Message Authentication Codes

module MAC

type text = bytes  val macsize

type key

type mac = b:bytes{Length(b)=macsize}

val GEN : unit -> key
val MAC : key -> text -> mac
val VERIFY: key -> text -> mac -> bool
module MAC

val macsize

val MAC : k:key -> t:text{Msg(k,t)} -> mac
val VERIFY: k:key -> t:text -> mac

val GEN : unit -> key

val MAC : k:key -> t:text{Msg(k,t)} -> mac

val VERIFY: k:key -> t:text -> mac

MAC keys are abstract

MACs are fixed sized

ideal F* interface

Msg is specified by protocols using MACs

“All verified messages have been MACed”
module MAC

open System.Security.Cryptography

let macsize = 20

let GEN() = randomBytes 16

let MAC k t = (new HMACSHA1(k)).ComputeHash t

let VERIFY k t m = (MAC k t = m)
Sample computational assumption:

**Resistance to Chosen-Message Existential Forgery Attacks (INT-CMA)**

```fsharp
module INT_CMA_Game
open Mac

let private k = GEN()
let private log = ref []
let mac t =
    log := t::!log
    MAC k t

let verify t m =
    let v = VERIFY k t m in
    if v && not (mem t !log) then FORGERY
    v
```

**Computational Safety**

A probabilistic polytime program calling `mac` and `verify` forges a MAC only with negligible probability.²

**CMA game (coded in F#)**
Computational Safety for MACs

ideal system

- Mac
  - F# interface
- Ideal filter
  - Ideal MAC
- RPC protocol
  - secure RPC

Any p.p.t. adversary
perfectly safe by typing

concrete system

- Mac
  - F# interface
- RPC protocol
  - sample protocol typed against ideal MAC interface
- Protocol adversary
  - typed against RPC interface

Any p.p.t. adversary
safe too, with probability $1 - 1/\varepsilon$

Concrete algorithm assumed INT-CMA computationally

error correction making VERIFY returns false on forgeries

INT-CMA adversary
Sample ideal functionality:

Supporting Key Compromise

```
module MAC

type text = bytes
val macsize

type key

val mac = b:bytes{Length(b)=macsize}

predicate Msg of key * text

val GEN : unit -> key
val MAC : k:key -> t:text{Msg(k,t)} -> mac
val VERIFY: k:key -> t:text -> mac
                  -> b:bool{ b=true => Msg(k,t)}

val keysize

val keybytes = b:bytes{Length(b)=keysize}

val LEAK:  k:key{!t. Msg(k,t)} -> b:keybytes
val COERCE: b:keybytes{...} -> k:key{...}
```

MAC keys are abstract

MACs are fixed sized

Msg is specified by protocols using MACs

“All verified messages have been MACed”

MAC keys have concrete representations

It is safe to turn keys into bytes when all messages are verifiable
Perfect Secrecy by Typing

- Secrecy is expressed using observational equivalences between systems that differ on their secrets
- We prove (probabilistic, information theoretic) secrecy by typing, relying on type abstraction

\[ I_\alpha = \alpha, \ldots, x : T_\alpha, \ldots \]

\[ P_\alpha \text{ range over pure modules such that } \vdash P_\alpha \rightsquigarrow I_\alpha. \]

**Theorem (Secrecy by Typing).**
Let \( A \) such that \( I_\alpha \vdash A : \text{bool} \).
For all \( P^0_\alpha \) and \( P^1_\alpha \), we have \( P^0_\alpha \cdot A \approx P^1_\alpha \cdot A \).
Plaintext Modules

- Encryption is parameterized by a module that abstractly define plaintexts, with interface

```plaintext
module Plaintext
val size: int
type plain
type repr = b:bytes{Length(b)=size}
val coerce : repr -> plain // turning bytes into secrets
val leak : plain -> repr // breaking secrecy!
val respond: plain -> plain // sample protocol code
```

If we remove the `leak` function, we get secrecy by typing

If we remove the `coerce` function, we get integrity by typing

Plain may also implement any protocol functions that operates on secrets
• Relying on basic cryptographic assumptions (IND-CPA, INT-CTXT) its ideal implementation never accesses plaintexts!

Formally, ideal AE is typed using an abstract plain type

```
module AE
open Plaintext

type key
type cipher = b:bytes{Length(b)= size + 16}

val GEN: unit -> key
val ENC: key -> plain -> cipher
val DEC: key -> cipher -> plain option
```

ENC k p encrypts instead zeros to c & and logs (k,c,p)
DEC k c returns Some(p) when (k,c,p) is in the log, or None
An Ideal Interface for CCA2-Secure Encryption

```ocaml
module PKENC
open Plain
val pksize: int
type skey
type pkey = b:bytes { PKey(b) \Æ }

val ciphersize: int
type cipher = b:bytes { Length(b) = ciphersize } 

val GEN: unit -> pkey * skey
val ENC: pkey -> plain -> cipher
val DEC: skey -> cipher -> plain
```

- Its **ideal implementation** encrypts zeros instead of plaintexts so it never accesses plaintext representations, and can be typed parametrically.
Typed Secrecy from CCA2-Secure Encryption

**Theorem 7** (Asymptotic Secrecy).
Let $P^0$ and $P^1$ p.p.t. secret with $\vdash P^b \sim I_{\text{PLAIN}}$.
Let $C_{\text{ENC}}$ p.p.t. CCA2-secure with $I_{\text{PLAIN}}^C \vdash C_{\text{ENC}} \sim I_{\text{ENC}}^C$.
Let $A$ p.p.t. with $I_{\text{PLAIN}}, I_{\text{ENC}} \vdash A : \text{bool}$.

$$P^0 \cdot C_{\text{ENC}} \cdot A \approx_{\varepsilon} P^1 \cdot C_{\text{ENC}} \cdot A.$$

**Theorem 8** (Ideal Functionality).
Let $P$ p.p.t. with $\vdash P \sim I_{\text{PLAIN}}^C$ (not necessarily secret).
Let $C_{\text{ENC}}$ p.p.t. CCA2-secure with $I_{\text{PLAIN}}^C \vdash C_{\text{ENC}} \sim I_{\text{ENC}}^C$.
Let $A$ p.p.t. with $I_{\text{PLAIN}}, I_{\text{ENC}} \vdash A$.

$$P \cdot C_{\text{ENC}} \cdot A \approx_{\varepsilon} P \cdot C_{\text{ENC}} \cdot F_{\text{ENC}} \cdot A.$$
Variants: CPA & Authentication

- With **CPA-secure encryption**, we have a **weaker** ideal interface that demands ciphertext integrity before decryption

  ```
  predicate Encrypted of key * cipher
  val ENC: k:key -> plain -> c:cipher{Encrypted(k,c)}
  val DEC: k:key -> c:cipher{Encrypted(k,c)} -> plain
  ```

- With **authenticated encryption**, we have a **stronger** ideal interface that ensures plaintext integrity (much as MACs)

  ```
  predicate Msg of key * plain // defined by protocol
  val ENC: k:key -> p:plain{Msg(k,p)} -> cipher
  val DEC: k:key -> cipher -> p:plain{Msg(k,p)} option
  ```
our main TLS API (outline)

Each application provides its own plaintext module for data streams:

• Typing ensures secrecy and authenticity at safe indexes

Each application creates and runs session & connections in parallel

• Parameters select ciphersuites and certificates

• Results provide detailed information on the protocol state

```scala
type cn // for each local instance of the protocol

// creating new client and server instances
val connect: TcpStream -> params -> (;Client) nullCn Result
val accept: TcpStream -> params -> (;Server) nullCn Result

// triggering new handshakes, and closing connections
val rehandshake: c:cn{Role(c)=Client} -> cn Result
val request: c:cn{Role(c)=Server} -> cn Result
val shutdown: c:cn -> TcpStream Result

// writing data
type (;c:cn,data:(;c) msg_o) ioresult_o =
| WriteComplete of c':cn
| WritePartial of c':cn * rest:(;c') msg_o
| MustRead of c':cn
val write: c:cn -> data:(;c) msg_o -> (;c,data) ioresult_o

// reading data
type (;c:cn) ioresult_i =
| Read of c':cn * data:(;c) msg_i
| CertQuery of c':cn
| Handshake of c':cn
| Close of TcpStream
| Warning of c':cn * a:alertDescription
| Fatal of a:alertDescription
val read : c:cn -> (;c) ioresult_i
```
Security theorem

Main crypto result: concrete TLS & ideal TLS are computationally indistinguishable

We prove that ideal miTLS meets its secure channel specification using standard program verification (typing)
Final Thoughts

Many pitfalls in cryptographic software
• Need to verify their design+implementation
• Need to verify crypto+protocol+application

Formal security proofs for real-world crypto protocols are now feasible
• TLS 1.3 is an ongoing successful experiment
• Similar results for SSH, Signal, etc.
• Many tools: ProVerif, CryptoVerif, F*, Tamarin, EasyCrypt, VST
• Try them out to build your next proof, or to implement your crypto protocols securely!
End of Part IV