Verified Implementations for Real-World Cryptographic Protocols

Karthikeyan Bhargavan

Inria

+ many co-authors at Inria, MSR, CMU, Stuttgart, ...

Collège de France, March 31 2022
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 !

Profitiez-en jusqu’au 31 mars!

Parlons de vos besoins
Is it safe to enter my pincode?
Secure Channels over Insecure Networks

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

The connection to this site is encrypted and authenticated using TLS 1.2, ECDHE_RSA with P-256, and AES_256_GCM.
Secure Channels in TLS [1994-2008]

\[ K = \text{KDF}(g^{xy}, [N_c, N_s]) \]

\[ \text{sign}(sk_s, [N_c, N_s, g^y]) \]

\[ g^x \]

\[ K = \text{KDF}(g^{xy}, [N_c, N_s]) \]

\[ \text{ae}(K, \text{data}) \]

\[ ... \]
Secure Channels in TLS [1994-2008]

Classic Two-Stage Protocol

1. **Authenticated Key Exchange**
   - Both parties compute a shared secret key $K$

2. **Authenticated Encryption**
   - Both parties exchange streams of data encrypted under $K$

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

```plaintext
K = KDF(g^{xy}, [N_c, N_s])
K = KDF(g^{xy}, [N_c, N_s])
sign(sk_s, [N_c, N_s, g^y])
g^x
BROWSER
WEBSITE
KDF(g^{xy}, [N_c, N_s])
```

$N_c$
Many Attacks on TLS Deployments [2011-16]

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

Crypto Weaknesses
• RC4, 3DES, MD5, PKCS#1 v1.5

Protocol Design Flaws
• Downgrade attacks, Transcript collisions

Implementation Bugs
• State machine bugs, Heartbleed

Often, a mix of all of the above!
Crypto Weaknesses
Diffie-Hellman Key Exchange

A

Knows $G = (g, p)$

$k = \text{kdf}(g^{xy} \mod p)$

B

Knows $G = (g, p)$

$k = \text{kdf}(g^{xy} \mod p)$

$g^x \mod p$

$g^y \mod p$
Diffie-Hellman Assumption

Security Assumption: An attacker who does not know $x$ or $y$ cannot compute $g^{xy}$ mod $p$
Weak Diffie-Hellman Groups

If the prime $p$ is too small, an attacker can compute the discrete log:

$$y = \log(g^y \mod p)$$

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

Current discrete log computation records:

- [Joux et al. 2005] 431-bit prime
- [Kleinjung et al. 2007] 530-bit prime
- [Bouvier et al. 2014] 596-bit prime
- [Kleinjung et al. 2017] 768-bit prime
- [Boudot et al. 2019] 795-bit prime
Real-World Diffie-Hellman Groups

Internet-wide scan of HTTPS servers (2015)
• 14.3M hosts, 24% support DHE
• 70,000 distinct groups \((p,g)\)

Many small-sized prime groups used for TLS
• 84% (2.9M) servers use 1024-bit primes
• 2.6% (90K) servers use 768-bit primes
• 0.0008% (2.6K) servers use 512-bit primes

Many servers support both strong and weak groups.
Protecting Protocols from Weak Crypto

Many deployed crypto algorithms are now considered weak
• RC4, MD5, 3DES, RSA-PKCS#1v1.5

The need for **backwards compatibility**
• Many systems cannot be updated frequently
• Need to continue support for legacy clients/servers

The benefits of **cryptographic agility**
• Gracefully transition from one algorithm to another
• Can already start supporting Post-Quantum algorithms
Protecting Protocols from Weak Crypto

Prove the security of protocols under weak assumptions
• Do you really need a collision-resistant hash function?
• Do you really need an IND-CCA secure encryption algorithm?

Analyze protocols that support both strong and weak crypto
• Prove security for connections that use strong crypto
• Show that strong crypto cannot be bypassed using weak crypto

Analyzing agile protocols by hand is too hard
• Large models, subtle assumptions and goals
• Need mechanized protocol verification tools
Protocol Design Flaws
Each crypto protocol composes a set of crypto constructions to achieve some target security goals

- TLS = DH + Sign + KDF + AE
- Each crypto algorithm may be individually strong, but they may not collectively achieve the desired security goal
Composing Sub-Protocols

Sequential or Vertical Composition
• Values generated by one protocol are used in the next
• e.g. Authenticated Key Exchange + Authenticated Encryption

Protocols with Algorithmic Agility
• Support for multiple algorithms within a single protocol
• e.g. allow weak and strong Diffie-Hellman groups

Parallel or Multi-mode Composition
• Many protocol flows to choose from
• Different sessions may choose a different modes
Agility: Diffie-Hellman Group Negotiation

Group Negotiation

\[ k = \text{kdf}(g^{xy} \mod p_{2048}) \]

Supports both strong and weak groups
Group Downgrade Attack

Computes the discrete log over the weak group to compute key

Removes Strong Groups
What went wrong?

Logjam Attack [2015]
• Cryptographic weakness: Weak Diffie-Hellman Groups
• Logical protocol flaw: Downgrade from Strong to Weak Group

Many other examples of downgrade+crypto attacks
• FREAK, SLOTH, DROWN, ...

These attacks only appear when analyzing complex composite protocol deployments
Implementation Bugs
Bugs in Protocol Implementations

Bugs when implementing cryptographic algorithms
• Functional correctness bugs, Side-channel leaks, ...
• e.g. Lucky13, Bleichenbacher, see OpenSSL CVEs

Bugs when parsing protocol messages and components
• Memory safety bugs (Heartbleed), Error propagation (Gotofail)
• X.509 certificate parsing errors (many CVEs)

Bugs in protocol state machine implementation (next)
• Allowing incorrect protocol flows (FREAK, SKIP)
Many possible protocol modes of TLS

Protocol versions
- TLS 1.2, TLS 1.1, TLS 1.0, SSLv3, SSLv2

Key exchanges
- ECDHE, FFDHE, RSA, PSK, ...

Authentication modes
- ECDSA, RSA signatures, PSK, ...

Authenticated Encryption Schemes
- AES-GCM, CBC MAC-Encode-Encrypt, RC4, ...

100s of possible protocol combinations!
State Machine for TLS-RSA Key Exchange

**ServerCertificate** \((m_3)\)
- \(\text{cert}(\text{pk}_S)\)

**ClientKeyExchange** \((m_4)\)
- \(\text{rsa-encrypt}(\text{pms}, \text{pk}_S)\)

**ClientFinished** \((m_5)\)
- \(\text{mac}(m_1-m_4, K)\)

**ServerFinished** \((m_6)\)
- \(\text{mac}(m_1-m_5, K)\)

**Server**
- \(\text{ClientHello}(v, \{kx_1, kx_2, \ldots\})\)
- \(\text{ServerHello}(v, kx = \text{RSA})\)
- \(\text{ServerCertificate}(\text{cert}_S)\)
- \(\text{ServerHelloDone}\)
- \(\text{ClientKeyExchange}(\text{rsaenc}(\text{pms}, \text{pk}_S))\)
- \(\text{ClientCCS}\)
- \(\text{ClientFinished}(\text{mac}(\log, \text{pms}))\)
- \(\text{ServerCCS}\)
- \(\text{ServerFinished}(\text{mac}^{'}(\log', \text{pms}))\)
- \(\text{ApplicationData}\)
State Machine for TLS-DHE Key Exchange

Client Key Exchange (m3)
- cert(pkS), rsa-sign(G | gy, skS)

Client Key Exchange (m4)
- gx

Client Finished (m5)
- mac(m1-m4, K)

Server Finished (m6)
- mac(m1-m5, K)

Client Hello (v, [kx1, kx2, ...])
- Server Hello (v, kx = DHE|ECDHE)
- Server Certificate (certS)
- Server Key Exchange (sign((G, g^y), skS))
- Server Hello Done
- Client Key Exchange (gx)
- Client CCS
- Client Finished (mac(log, g^wy))
- Server CCS
- Server Finished (mac(log', g^wy))
- Application Data
Composing Protocol State Machines

ClientHello($v, [kx_1, kx_2, \ldots]$)
  \rightarrow RSA

ServerHello($v, kx = RSA$)

ServerCertificate($cert_S$)

ServerHelloDone

ClientKeyExchange($rsaEnc(pms, pk_S)$)

ClientCCS

ClientFinished($mac/log, pms)$

ServerCCS

ServerFinished($mac/log, pms$)

ApplicationData

\[ + \]

ClientHello($v, [kx_1, kx_2, \ldots]$)

ServerHello($v, kx = DHE|ECDHE)$

ServerCertificate($cert_S$)

ServerKeyExchange($sign((G, g^p), sk_S)$)

ServerHelloDone

ClientKeyExchange($g^e$)

ClientCCS

ClientFinished($mac/log, g^e$)

ServerCCS

ServerFinished($mac/log, g^e$)

ApplicationData

\[ = \]

ClientHello($v, [kx_1, kx_2, \ldots]$)

ServerHello($v, kx = RSA$)

ServerCertificate($cert_S$)

ServerKeyExchange($\ldots$)

ServerHelloDone

ClientKeyExchange($\ldots$)

ClientCCS

ClientFinished($mac/log, \ldots$)

ServerCCS

ServerFinished($mac/log, \ldots$)

ApplicationData
Commonly Deployed TLS State Machine

RSA + DHE + ECDHE
+ Session Resumption
+ Client Authentication

- Covers most features used on the Web
Full SSL/TLS State Machine

- Fixed_DH
- DH_anon
- PSK
- SRP
- Kerberos
- *_EXPORT
- ...

These are all the ones implemented in OpenSSL
Testing TLS State Machines

Do popular TLS libraries conform to this state machine spec?


We built a fuzzing framework

- FlexTLS, based on miTLS, a verified implementation of TLS
- Generates non-conforming traces from a formal state machine spec
- Tests open-source libraries
Many, Many Bugs

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
Incorrectly Composing State Machines

- **RSA**
  - ClientHello($v, [kx_1, kx_2, \ldots]$)
  - ServerHello($v, kx = \text{RSA}$)
  - ServerCertificate($cert_S$)
  - ServerHelloDone
  - ClientKeyExchange($\text{rsaenc}(pms, pk_S)$)
  - ClientCCS
  - ClientFinished($\text{mac}(log, pms)$)
  - ServerCCS
  - ServerFinished($\text{mac}(log', pma)$)
  - ApplicationData\textsuperscript{*}

- **(EC)DHE**
  - ClientHello($v, [kx_1, kx_2, \ldots]$)
  - ServerHello($v, kx = \text{DHE|ECDHE}$)
  - ServerCertificate($cert_S$)
  - ServerKeyExchange($\text{sign}((G, g^x), sk_S)$)
  - ServerHelloDone
  - ClientKeyExchange($g^x$)
  - ClientCCS
  - ClientFinished($\text{mac}(log, g^{xy})$)
  - ServerCCS
  - ServerFinished($\text{mac}(log', g'^{xy})$)
  - ApplicationData\textsuperscript{*}

- Not equal

- **RSA**
  - ClientHello($v, [kx_1, kx_2, \ldots]$)
  - ServerHello($v, kx = \text{RSA}$)
  - ServerCertificate($cert_S$)
  - ServerHelloDone
  - ClientKeyExchange($\text{rsaenc}(pms, pk_S)$)
  - ClientCCS
  - ClientFinished($\text{mac}(log, pms)$)
  - ServerCCS
  - ServerFinished($\text{mac}(log', pma)$)
  - ApplicationData\textsuperscript{*}

- **(EC)DHE**
  - ClientHello($v, [kx_1, kx_2, \ldots]$)
  - ServerHello($v, kx = \text{DHE|ECDHE}$)
  - ServerCertificate($cert_S$)
  - ServerKeyExchange($\text{sign}((G, g^x), sk_S)$)
  - ServerHelloDone
  - ClientKeyExchange($g^x$)
  - ClientCCS
  - ClientFinished($\text{mac}(log, g^{xy})$)
  - ServerCCS
  - ServerFinished($\text{mac}(log', g'^{xy})$)
  - ApplicationData\textsuperscript{*}
Incorrectly Composing State Machines

Follows Postel’s robustness principle
• "Be conservative in what you do, be liberal in what you accept from others” (BAD for security!)

Introduces unexpected cases at the client
• Server skips ServerKeyExchange in DHE
• Server sends ServerKeyExchange in RSA

Correct clients should reject these cases
• Otherwise, they are not executing TLS anymore, and lose all its security guarantees
Network attacker impersonates api.paypal.com to a JSSE client

1. Send PayPal’s cert
Network attacker impersonates api.paypal.com to a JSSE client
1. Send PayPal’s cert
2. SKIP ServerKeyExchange (bypass server signature)
3. SKIP ServerHelloDone
Network attacker impersonates api.paypal.com to a JSSE client

1. Send PayPal’s cert
2. SKIP ServerKeyExchange (bypass server signature)
3. SKIP ServerHelloDone
4. SKIP ServerCCS (bypass encryption)
5. Send ServerFinished using uninitialized MAC key (bypass handshake integrity)
6. Send ApplicationData (unencrypted) as S.com
State Machine Attacks

Impact of SKIP on Java TLS Clients

- A network attacker can impersonate *any* server (Paypal, Amazon, Google) to *any* Java TLS client
- Affects all versions of Java until Jan 2015 CPU

Many other State Machine bugs in TLS libraries

- **FREAK**: combines crypto weakness, protocol flaw, and implementation bug
Recap: What goes wrong in TLS Deployments?

Crypto Weaknesses
• RC4, 3DES, MD5, PKCS#1 v1.5

Protocol Design Flaws
• Downgrade attacks, Transcript collisions

Implementation Bugs
• State machine bugs, Heartbleed

Often, a mix of all of the above!
How do we fix it?

Verify protocol software
• Verified Crypto Library (HACL*)
• Verified Protocol Code (DY*)

Today:
• Motivating Example: HACL*
• Brief taste of verification in F*
Reading Materials

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

HACL*
Towards High-Assurance Crypto Libraries

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)

```
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)

```c
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 * "h" p, where "h" 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);
}
```

Optimized 32-bit Code
a lot more code, with possible carry propagation bugs, or buffer overflows, or timing leaks.
Modular Arithmetic in Crypto

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

\[ 0 < x, y < p \]
\[ g \text{ fixed} \]

**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

\[
\begin{array}{c}
1101 \\
\times \quad 1010 \\
\hline
0000 \\
1101 \\
\downarrow \text{carry} \\
0000 \\
\hline
1101 \\
\hline
10000010
\end{array}
\]

\[
\begin{array}{c}
13 \\
\times \quad 10 \\
\hline
= 130
\end{array}
\]
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!
Optimizations in Modular Arithmetic

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

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 + carry should be r2 += 0 + 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, ...
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

[Zinzindohoué 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
HACL* Verification workflow

- **Sequences (Pure F*)**
- **Int (ℤ) (Pure F*)**
- **MachineInt (Pure F*)**
- **Buffers (Low*)**

**F* Standard Library**

- Secret Independence
- Functional Correctness
- Memory Safety

**HACL* Crypto Library**

- Port

- **Spec (Pure F*)**

**Standard (RFC, NIST)**

- **Transcribe**

**Verify (F*)**

- **success**
- **failure**
  - Potential memory safety bug, or functional correctness bug, or side-channel leak.

**Compile (KreMLin)**

- **success**
- **failure**
  - Source code not in Low*; Cannot be compiled to C.

**Verified Code (C)**
Example: Poly1305 MAC Algorithm

Poly1305 is a message authentication code

\[ poly(k, m, w_1...w_n) = m + w_1k^1 + ... + w_nk^n \mod (2^{130} - 5) \]

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
Efficiently Implementing Poly1305

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)

```
val fsum: 
a:felem →  
b:felem →  
Stack unit  
(requires (\lambda h → live h a ∧ live h b ∧ disjoint a b ∧ no_overflow h a b len))
(ensures (\lambda h₀ → h₁ → live h₁ a ∧ live h₁ b ∧ modifies_1 a h₀ h₁ ∧ eval h₁ a = eval h₀ a + eval h₀ b))
```
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)

abstract type hint64_t
val v: hint64_t → GTot uint64_t

val (+): hint64_t → hint64_t → Tot hint64_t
val (^): hint64_t → hint64_t → Tot hint64_t
// val (/): hint64_t → hint64_t → Tot hint64_t

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

Standard (RFC, NIST)

Spec (Pure 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
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
Low* Code for Poly1305

Compiled C Code
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-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
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>-</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 64x64→128 multiplications but still faster because of less carry propagation

reduce modulo $2^{255} - 19$
An instruction set with 2 carry flags can significantly reduce the cost of carry propagation!

**Intel(R) Xeon(R) CPU E3-1505M v5 @ 2.80GHz**
- donna64: 160942 cycles per call
- hac164: 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**
- amd64: 143628 cycles per call
- fiat32: 307971 cycles per call
- donna32: 544254 cycles per call

**New speed record?**
Measurements using Jason Donenfeld’s Linux Kernel Benchmarking Suite for WireGuard.
Saturated 64-bit Arithmetic with Intel ADX

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 2^p 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]
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

```
mul2:
push %r12
push %r13
push %r14
mov %rdx, %rcx
movq 0(%rsi), %rdx
mulxq 0(%rcx), %r8, %r9
xor %r10, %r10
movq %r8, 0(%rdi)
mulxq 8(%rcx), %r10, %r11
adox %r9, %r10
movq %r10, 8(%rdi)
mulxq 16(%rcx), %r12, %r13
adox %r11, %r12
mulxq 24(%rcx), %r14, %rdx
adox %r13, %r14
mov $0, %rax
adox %rdx, %rax
movq 8(%rsi), %rdx
```
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></td>
<td></td>
</tr>
<tr>
<td>ChachaPoly</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>Hashes</td>
<td></td>
<td></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>SHA-EXT (for SHA2-224+SHA2-256)</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>agile over hash</td>
</tr>
<tr>
<td>HKDF</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>ECC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Curve25519</td>
<td>yes</td>
<td>BMI2 + ADX</td>
</tr>
<tr>
<td>Ciphers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chacha20</td>
<td>yes</td>
<td>AES NI + AVX</td>
</tr>
<tr>
<td>AES128, 256</td>
<td></td>
<td>AES NI + AVX</td>
</tr>
<tr>
<td>AES-CTR</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### 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

Idea 2: Write & verify generic SIMD code
- Compiles to platform-specific code
- A single target for formal verification
- A basepoint for further optimization

- Hard target for formal verification
  (probably also awful for maintenance)
## HACL* Vectorization Performance

### CHACHA20

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

### POLY1305

<table>
<thead>
<tr>
<th>32-bit Scalar</th>
<th>1.5 cy/b</th>
</tr>
</thead>
<tbody>
<tr>
<td>128-bit Vectorized (AVX)</td>
<td>0.75 cy/b</td>
</tr>
<tr>
<td>256-bit Vectorized (AVX2)</td>
<td>0.39 cy/b</td>
</tr>
<tr>
<td>Fastest Assembly (OpenSSL AVX2)</td>
<td>0.34 cy/b</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>hacspect</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>hacspect</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* Code in Deployment

WireGuard

Security

Focus on

Upgrades

Built on

Simple & Easy-to-use

Tezos is an open-source smart contract blockchain focused on upgrades and built on top of Simple & Easy-to-use. WireGuard aims to be an extremely lightweight and high-speed variant of the well-known tunneling protocol, aiming to be used as a fast, simple, and easy-to-use way to build a high-speed, secure, and easy-to-use VPN for running on small systems and widely deployable. It is a lightweight, secure, easy-to-use, and widely deployable. It is a lightweight, secure, and easy-to-use, and widely deployable.

### Simple & Easy-to-use

Tezos is an open-source smart contract blockchain focused on upgrades and built on top of Simple & Easy-to-use. WireGuard aims to be an extremely lightweight and high-speed variant of the well-known tunneling protocol, aiming to be used as a fast, simple, and easy-to-use way to build a high-speed, secure, and easy-to-use VPN for running on small systems and widely deployable. It is a lightweight, secure, easy-to-use, and widely deployable. It is a lightweight, secure, easy-to-use, and widely deployable.

Electronics

open source

software development kit (SDK)

SDK that makes voting more secure, transparent and accessible. Announced on at the Build developer conference, ElectionGuard enables end-to-end verification of elections as well as support the publication of results from ballot comparison audits. The ElectionGuard SDK leverages homomorphic encryption to ensure that votes recorded by electronic systems of any type remain encrypted, secure, and secret. Results can be published online or made available to third-party organizations for secure validation, and allow individual voters to confirm their votes were correctly counted.

Open-Source

This library and all linked ElectionGuard projects, are licensed under the MIT license. There is no fee for using ElectionGuard.

Getting Started

ElectionGuard is always improving. To keep up with the latest, check our official site on GitHub Pages and our roadmap. For those looking to get started, we recommend the following resources.
Conclusions

End-to-End verification of protocol stacks is now feasible
• Using proof-oriented programming languages like F*
• Crypto (HACL*), Parsers (EverParse), Protocols (miTLS, DY*)

Area is still maturing and is under active research
• Verifying optimized low-level code in C and assembly
• Verifying protocols that use ZK proofs, or MPC, or FHE, or PQ

Many open problems for future work
• Proving the absence of side channel attacks
• Verifying code written by non-verification experts