HACL* in Mozilla Firefox

Formal methods and high assurance applications for the web

B. Beurdouche
K. Bhargavan
J. Protzenko
J-K. Zinzindohoué
(Project Everest)

F. Kiefer
E. Rescorla
T. Taubert
M. Thomson
(Mozilla)

Real World Crypto 2018
Let’s focus on Crypto[graphy]!
Implementing cryptography is difficult

**Memory Safety**
(think Heartbleed)

**Side channels**
(think Lucky 13)

**Functional correctness**
Functional correctness is difficult

[2016] Integer overflow in OpenSSL’s Poly1305

```c
/* last reduction step: */

/* a) h2:h0 = h2<<128 + d1<<64 + d0 */

h0 = (u64)d0;

h1 = (u64)(d1 += d0 >> 64);

h2 += (u64)(d1 >> 64);

/* b) (h2:h0 += (h2:h0>>130) * 5) %= 2^130 */

c = (h2 >> 2) + (h2 & ~3UL);

h2 &= 3;

h0 += c;

h1 += (c = CONSTANT_TIME_CARRY(h0,c)); /* doesn’t overflow */
```
Implementing is hard for everyone.

Even for very skilled programmers or cryptographers!
Network Security Services (NSS) library

Multi product security library
• Joint effort from Mozilla, RedHat...
• Security Library for Firefox in C/C++
• Used in RHEL, Fedora, BSDs...

Large number of primitives
• Both recent and legacy primitives for interoperability

Higher level components
• Protocols (TLS...)
• Cryptographic APIs (WebCrypto, PKCS...)
Redesigning NSS

“NSS is old, there is a lot of legacy code”

“How can we make NSS more modern and get higher confidence in its correctness?”

There was no clear way on how to get there...
- Clean room redesign “à la BoringSSL”
- More money ?! More hiring ?!

Decision
- Improve step-by-step the confidence in code correctness using formal verification
Research challenge from the OpenSSL team

How can the community help?

- Formal verification of crypto code
  - Hitting $< 2^{-64}$ corner cases with unit testing is difficult.
  - New-ish elliptic curve implementations: P-224, P-256, P-521 - fast and constant-time. But are they correct?
  - Regression testing (again!) for bug attacks and oracle attacks.

Emilia Kasper, Real World Crypto (2015)
Formal methods inbound

Recent academic developments for Cryptography

Verifying Curve25519 Software

Ing-Chen¹, Chang-Hong Hsu², Hsin-Hung Lin³, Peter Schwabe⁴, MiBow-Yaw Wang¹, Bo-Yin Yang¹, and Shang-Yi Yang¹

¹ Institute of Information Science
² Academia Sinica
³ 128 Section 2 Academia Road, Taipei 115-29, Taiwan

Verifiable side-channel security of cryptographic implementations: constant-time MEE-CBC

José Bacelar Almeida¹², Manuel Barbosa¹³, Gilles Barthe⁴, and François Dupressoir⁴

¹ HASlab – INESC TEC
² University of Minho
³ DCC-FC, University of Porto
⁴ IMDEA Software Institute


Verification of a Cryptographic Primitive: SHA-256

ANDREW W. APPEL, Princeton University

Verified correctness and security of OpenSSL HMAC

To appear in 24th Usenix Security Symposium, August 12, 2015

Lennart Beringer
Princeton Univ.

Adam Petcher
Harvard Univ. and
Princeton Univ.

Katherine Q. Ye
Princeton Univ.

Andrew W. Appel
Princeton Univ.

Verifying Constant-Time Implementations

José Bacelar Almeida
HASLab - INESC TEC & Univ. Minho

Manuel Barbosa
HASLab - INESC TEC & DCC FCUP

Gilles Barthe
IMDEA Software Institute

François Dupressoir
IMDEA Software Institute

Michael Emmi
Bell Labs, Nokia
What kind of verification and how?

Assembly, C or High-Level Languages?

Code generation or Verification of existing code?
HACL*: A Verified Modern Cryptographic Library

Jean Karim Zinzindohoué
INRIA

Jonathan Protzenko
Microsoft Research

Karthikeyan Bhargavan
INRIA

Benjamin Beurdouche
INRIA

ABSTRACT

HACL* is a verified portable C cryptographic library that implements modern cryptographic primitives such as the ChaCha20 and Salsa20 encryption algorithms, Poly1305 and HMAC message authentication, SHA-256 and SHA-512 hash functions, the Curve25519 elliptic curve, and Ed25519 signatures.

HACL* is written in the F* programming language and then compiled to readable C code. The F* source code for each cryptographic primitive is verified for memory safety, mitigations against timing side-channels, and functional correctness with respect to a succinct high-level specification of the primitive derived from its published standard. The translation from F* to C preserves these properties and the generated C code can itself be compiled via the CompCert verified C compiler or mainstream compilers like GCC or CLANG. When compiled with GCC on 64-bit platforms, our primitives are as fast as the fastest pure C implementations in OpenSSL and libsodium, significantly faster than the reference C code in TweetNaCl, and between 1.1x-5.7x slower than the fastest hand-optimized vectorized assembly code in SUPERCOP.

HACL* implements the NaCl cryptographic API and can be used as a drop-in replacement for NaCl libraries like libsodium and

the absence of entire classes of potential bugs. In this paper, we will show how to implement a cryptographic library and prove that it is memory safe and functionally correct with respect to its published standard specification. Our goal is to write verified code that is as fast as state-of-the-art C implementations, while implementing standard countermeasures to timing side-channel attacks.

A Library of Modern Cryptographic Primitives. To design a high-assurance cryptographic library, we must first choose which primitives to include. The more we include, the more we have to verify, and their proofs can take considerable time and effort. Mixing verified and unverified primitives in a single library would be dangerous, since trivial memory-safety bugs in unverified code often completely break the correctness guarantees of verified code. General-purpose libraries like OpenSSL implement a notoriously large number of primitives, totaling hundreds of thousands of lines of code, making it infeasible to verify the full library. In contrast, minimalist easy-to-use libraries such as NaCl [17] support a few carefully chosen primitives and hence are better verification targets. For example, TweetNaCl [19], a portable C implementation of NaCl is fully implemented in 700 lines of code.

For simplicity on presentation, cryptographic libraries in

C are not typically verified, and verification is often left to the

library implementers. However, this approach is not always practical or possible. The increasing complexity of modern cryptographic algorithms makes it difficult to trust the correctness of their implementations, and verification is becoming more and more important. In this paper, we will present a verified library that implements modern cryptographic primitives, and we will show how to implement a library that is both verified and fast.
F* verification workflow

Crypto Standard (RFC, NIST...)
Spec (F*)
Verify (F*)
Compile (KreMLin)
Verified Code (C)

Trusted Library (F*)
Code (F*)

Memory safety
Functional correctness
Secret independence

Correctness theorem [ICFP2017]

Potential bug
Cannot be compiled to C
Formal verification can scale up!

Functionalities

- Hash function (SHA-2)
- Message authentication (HMAC, Poly1305)
- Symmetric ciphers (Chacha20, Salsa20)
- Key Exchange algorithm (Curve25519)
- Signature scheme (Ed25519)
- AEAD (Chacha20Poly1305)

<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>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
module Spec.Poly1305

(* Field types and parameters *)
let prime = pow2 130 - 5

(* Specification code *)
let encode (w:word) =
  (pow2 (8 * length w)) `fadd` (little_endian w)

let rec poly (txt:text) (r:e:elem) : Tot elem (decreases (length txt)) =
  if length txt = 0 then zero
  else
    let a = poly (Seq.tail txt) r in
    let n = encode (Seq.head txt) in
    (n `fadd` a) `fmul` r

let encode_r (rb:word_16) =
  (little_endian rb) &\| 0x0ffffffc0ffffffc0ffffffc0ffffff

let finish (a:elem) (s:word_16) : Tot tag =
  let n = (a + little_endian s) % pow2 128 in
  little_bytes 16ul n

let rec encode_bytes (txt:bytes) : Tot text (decreases (length txt)) =
  if length txt = 0 then createEmpty
  else
    let w, txt = split txt (min (length txt) 16) in
    append_last (encode_bytes txt) w

let poly1305 (msg:bytes) (k:key) : Tot tag =
  let text = encode_bytes msg in
  let r = encode_r (slice k 0 16) in
  let s = slice k 16 32 in
  finish (poly text r) s
How does the stateful code and proofs look like?
Low* Poly1305 compiled to C

```c
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)0xfffffffff;
    uint64_t r0 = a0 >> (uint32_t)44;
    uint64_t a1 = (a10 + r0) & (uint64_t)0xfffffffff;
    uint64_t r1 = (a10 + r0) >> (uint32_t)44;
    uint64_t a2 = a20 + r1;
    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 t0 = t0 & (((uint64_t)1) << ((uint32_t)44) - (uint64_t)1);
    uint64_t t1 = t1 + (t0 >> (uint32_t)44);
    acc[0] = t0;
    acc[1] = t1;
    uint64_t a00 = acc[0];
    uint64_t a11 = acc[1];
    uint64_t a22 = acc[2];
    uint64_t t0 = FStar.UInt64_gte_mask(a00, (uint64_t)0xfffffffff);
    uint64_t t1 = FStar.UInt64_eq_mask(a11, (uint64_t)0xfffffffff);
    uint64_t t2 = FStar.UInt64_eq_mask(a22, (uint64_t)0x100000000);
    uint64_t mask0 = mask0 & mask1 & mask2;
    uint64_t mask1 = (uint64_t)0xfffffffff;
    uint64_t mask2 = (uint64_t)0x100000000;
    HACL_Impl_Poly1305_64_poly1305_last_pass_spec((uint64_t)0x40);
}
```

Low* code proven by Coq
HACL* in Mozilla Firefox
HACL* in Mozilla Firefox

Firefox 57 "Quantum" was a major release for Mozilla
  • Includes verified cryptography from HACL* (Curve25519)

Firefox Nightly already has more
  • Chacha20 and Poly1305

Next batch of primitives on its way
  • Vectorized Chacha20Poly1305 + Ed25519
  • SHA2 + HMAC + HKDF
  • RSA_PSS + P256 ...
How does one go from an academic project to production code in the industry?
Integration process constraints

Performance
  • Reducing performance is not acceptable (in general)

Code integration
  • Readable, reviewable code

Toolchain integration
  • Insert verification into the current dev. workflow

Deployment and support
  • NSS runs on almost everything
  • API and ABI stability
## HACL* Performance (C code)

### CPU cycles/byte

Lower is better

<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><strong>Curve25519</strong></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 NSS code
Code review (Phabricator)

```c
if (i == (uint32_t)0) {
} else {

    ekr

    WAT?

    uint32_t i_ = i - (uint32_t)1;
    Hacl_EC_Ladder_SmallLoop_cmult_small_loop_double_step(nq, nqpq, nq2, nqpq2, q, byt);
    uint8_t byt_ = byt << (uint32_t)2;
    Hacl_EC_Ladder_SmallLoop_cmult_small_loop(nq, nqpq, nq2, nqpq2, q, byt_, i_);
}
```

Removing empty branches, unreachable code...
Improving code quality

Better variable naming
Removing intermediate variables
HACL* verification toolchain in NSS CI (treeherder)
Supporting multiple platforms

Large number of supported platforms

• CI does not support all platforms

  Bug 1396301
  verified/kremlib.h:204:23: error: implicit declaration of function 'le64t
  declaration]
  RESOLVED FIXED

• Trusted code base is a problem

  Bug 1419009
  Sigsegv at Hacl_EC_crypto_scalarmult on Solaris
  RESOLVED INVALID

• Some bugs can be introduced by contributors

  Bug 1405268
  shlibsing fails on Solaris due missing htole64 symbol
  RESOLVED DUPLICATE of bug 1420060
A common workflow

1. Write F* spec & code
   - success
   - Prove Low* code
     - failure
     - Extract to C and Test
       - success
       - Verified Code (C)
         - failure
         - Format and Audit
           - failure
           - CI Verification and Tests
             - success
             - Production
               - success

NSS integration tasks #20

- Production branch for NSS based on recent HACL/J/NIKOSIN, master branches
- Export HACL/NIKOSIN tests to NSS
- Setup the NSS-CI based on the HACL-Decker image
- Identify a set of working F*/NIKOSIN versions working as expected
- Rename bundles to be prefixed with -nick_ (NSS fails because cached.h already exists)
- Reduce plugin code base from 1.1kahuku
- Remove dependency into C-netsuite and Firefox
- Generate new snapshot with parentheses to silence -warner
- Using verified integers
- NSS-CI-Decker image
- Some void functions have returns (not all). Can we not do that? (@franziskuskieler)
- Remove code extraction artifacts (see GnuPro20 below)

Improvements

- Get rid of the patches in the production branch (466)
- Automatic generation of the nickname prefix using Bundles (465)
- Remove dependencies to testlib automatically from generated-header files (469)
- Cleanup headers by using private in the F* code and make NIKOSIN extract in .c files instead
- Generate correct keywords from NIKOSIN
- Various code generation improvements (see NIKOSIN-20)

Future primitives

- Curve25519 (32bits) through the 115bit version
- SHA2/MAAC-HKDF (incremental with strong interface)
- RSA-PSS (6 Generic Signum)
- P256
- AES [ref] + AES-NI

Licensing and Headers

- Waiting on logio to see if Apache2 is possible. Apache2 is OK for NSS
- Copyright header on the C-code
What’s next?

The future of NSS
• Removing more obsolete code
• Mixing-in other formal methods
• Integrate formally verified assembly
• Verifying parsers and protocols

The future of HACL*
• Implement new primitives
• Reduce proof effort and verification time
• Reduce trust in our tools (verify KreMLin, F*...)
• Support more platforms (WASM, RIOT...)
Use it ! Test it ! Break it !

(NSS crypto is eligible to Mozilla’s bug bounty program)

Get in touch !

@beurdouche
benjamin.beurdouche@inria.fr