Verified low-level programming embedded in $F^*$

Jonathan Protzenko
Jean-Karim Zinzindohoué
Aseem Rastogi
Tahina Ramananandro
Peng Wang
Santiago Zanella-Béguelin
Antoine Delignat-Lavaud
Cătălin Hrițcu
Karthikeyan Bhargavan
Cédric Fournet
Nikhil Swamy

Microsoft Research
INRIA Paris
Microsoft Research
Microsoft Research
MIT CSAIL
Microsoft Research
Microsoft Research
INRIA Paris
Microsoft Research
Microsoft Research
Everest:
Deploying Verified-Secure Implementations in the HTTPS Ecosystem
Within HTTPS: the TLS protocol

TLS stands for *transport layer security*.

TLS is made of up of two halves:
- the *protocol* layer
- the *record* layer

Specifically, the *record* layer contains the *cryptographic routines*. 
A sample cryptographic operation: Poly1305

Poly1305 is a message authentication code.

\[
MAC(k, m, \vec{w}) = m + \sum_{i=1}^{\mid\vec{w}\mid} w_i \times k^i
\]

It authenticates the data \(\vec{w}\) by:

- encoding it as a polynomial in the prime field \(2^{130} - 5\)
- evaluating it at a random point \(k\) (first part of the key)
- masking the result with \(m\) (second part of the key)
A sample cryptographic operation: Poly1305

Poly1305 is a message authentication code.

\[ MAC(k, m, \vec{w}) = m + \sum_{i=1}^{\vec{w}} w_i \times k^i \]

A typical 64-bit arithmetic implementation:

- represents elements of the prime field \((p = 2^{130} - 5)\) using three limbs holding 42 + 44 + 44 bits in 64-bit registers
- uses \((a \times 2^{130} + b)\%p = (a + 4a + b)\%p\) for reductions
- unfolds the loop
A simple cryptographic operation: Poly1305?

These heavily optimized C implementations have bugs.
A simple cryptographic operation: Poly1305?

OpenSSL Security Advisory [10 Nov 2016]

ChaCha20/Poly1305 heap-buffer-overflow (CVE-2016-7054)

Severity: High

TLS connections using *-CHACHA20-POLY1305 ciphersuites are susceptible to a DoS attack by corrupting larger payloads. This can result in an OpenSSL crash. This issue is not considered to be exploitable beyond a DoS.
A simple cryptographic operation: Poly1305?

These heavily optimized C implementations have bugs.

OpenSSL Security Advisory [10 Nov 2016]
__________________________________________________________
ChaCha20/Poly1305 heap-buffer-overflow (CVE-2016-7054)
__________________________________________________________
Severity: High

TLS connections us attack by corrupt issue is not consi

[openssl-dev] [openssl.org #4482] Wrong results with Poly1305 functions

Hanno Boeck via RT rt at openssl.org
Fri Mar 25 12:10:32 UTC 2016

- Previous message: [openssl-dev] [openssl.org #4480] PATCH: Ubuntu 14 (x86_64): Compile errors and warnings when using "no-asm -ansi"
- Next message: [openssl-dev] [openssl.org #4483] Re: [openssl.org #4482] Wrong results with Poly1305 functions
- Messages sorted by: [date] [thread] [subject] [author]

Attached is a sample code that will test various inputs for the Poly1305 functions of openssl.

These produce wrong results. The first example does so only on 32 bit, the other three also on 64 bit.
A simple cryptographic operation: Poly1305?

OpenSSL Security Advisory [10 Nov 2016]
---------------------------------------------------------------------
ChaCha20/Poly1305 heap-buffer-overflow (CVE-2016-7054)
---------------------------------------------------------------------
Severity: High

TLS connections us
attack by corrupt
issue is not cons

openssl-dev] [openssl.org #4482] Wrong results with Poly1305 functions

Hanno Boeck via RT rt at openssl.org
Fri Mar 25 12:10:32 UTC 2016
• Previous message: [openssl-dev] [openssl.org #4439] poly1305-x86.pl produces incorrect output
• Next message: [openssl-dev] [openssl.org #4439] poly1305-x86.pl produces incorrect output
• Messages sorted by: [author] [subject] [date] [thread]

Hi folks,
You know the drill. See the attached poly1305_test2.c.

$ OPENSSL_i32cap=0 ./poly1305_test2
PASS
$ ./poly1305_test2
Poly1305 test failed.
got: 2637408fe03086ea73f971e3425e2820
expected: 2637408fe13086ea73f971e3425e2820

I believe this affects both the SSE2 and AVX2 code. It does seem to be dependent on this input pattern.

This was found because a run of our SSL tests happened to find a problematic input. I've trimmed it down to the first block where they disagree.

I'm probably going to write something to generate random inputs and stress all your other poly1305 codepaths against a reference.

J. Protzenko et al. — ICFP'17
Specifying, programming and verifying Poly1305

module Spec.Poly1305

let prime = pow2 130 - 5

type elem = e:Z{e ≥ 0 ∧ e < prime}

let fadd (e1:elem) (e2:elem) = (e1 + e2) % prime

let fmul (e1:elem) (e2:elem) = (e1 × e2) % prime

(* 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

Auto-saving... done
Verified low-level programming embedded in F*
The design of Low*
High-level verification for low-level code

For code, the programmer:
- opts in the Low* effect to model the C stack and heap;
- uses low-level libraries for arrays and structs;
- leverages combinator libraries to get C loops;
- meta-programs first-order code;
- relies on data types sparingly.

For proofs and specs, the programmer:
- can use all of F*,
- prove memory safety, correctness, crypto games, relying on
- erasure to yield a first-order program.

Motto: the code is low-level but the verification is not.
Our low-level, stack-based memory model.

```plaintext
effect Stack (a:Type) (pre:st_pre) (post: (mem -> Tot (st_post a))) =
STATE a (fun (p:st_post a) (h:mem) ->
    pre h \<= (\forall a h1.
        (pre h \<= post h a h1 \<= equal_domains h h1) \implies p a h1))

let equal_domains (m0:mem) (m1:mem) =
    m0.tip = m1.tip
    \<= Set.equal (Map.domain m0.h) (Map.domain m1.h)
    \<= (forall r. Map.contains m0.h r \implies
        Heap.equal_dom (Map.sel m0.h r) (Map.sel m1.h r))
```

Preserves the layout of the stack and doesn’t allocate in any caller frame.
Our **low-level, stack-based** memory model.

\[
\text{effect Stack (a:T) => (pre pre a post post a)) =}
\text{STATE a (fun (p:pre a) => (post a))}
\text{pre h \ (\forall a h1.}
\text{ (pre h \ post h a h1 \ equal_domains h h1) => p a h1))}
\]

\[
\text{let equal_domains (m0:mem) (m1:mem) =}
\text{ m0.tip = m1.tip}
\text{ Set.equal (Map.domain m0.h) (Map.domain m1.h)}
\text{ Set.equal (Map.sel m0.h r) (Map.sel m1.h r))}
\]

Preserves the **layout** of the stack and **doesn’t allocate** in any caller frame.
High-level verification for low-level code (2)

Our low-level, stack-based memory model.

effect Stack (a:Type) (pre:st_pre) (post: (mem -> Tot (st_post a))) = STATE a (fun (p:st_post a) (h:mem) ->
    pre h /
    (\ a h1. (pre h /
    post h a h1 /
    equal_domains h h1) ==> p a h1))

let equal_domains (m0:mem) (m1:mem) =
    m0.tip = m1.tip
    \ Set.equal (Map.domain m0.h) (Map.domain m1.h)
    \ (forall r. Map.contains m0.h r =>
    Heap.equal_dom (Map.sel m0.h r))

the tip remains the same

Preserves the layout of the stack and doesn’t allocate in any caller frame.
Our low-level, sequence-based buffer model.

```fsharp
val index: #a:Type -> b:buffer a -> n:UInt32.t{v n < length b} ->
  Stack a
  (requires (fun h -> live h b))
  (ensures (fun h0 z h1 -> live h0 b \ h1 == h0
        \ z == Seq.index (as_seq h0 b) (v n)))
let index #a b n =
  let s = !b.content in
  Seq.index s (v b.idx + v n)
```

We swap this F* model with a low-level implementation. 
`buffer int` becomes `int*` and `index b i` becomes `b[i]`. 
Our low-level, sequence-based buffer model.

```f
val index : #a : Type -> b : buffer a -> n : UInt32.t { v n < length b } ->
  Stack a
  (requires (fun h -> live h b))
  (ensures (fun h0 z h1 -> live h0 b \ h1 == h0
    \ z == Seq.index (as_seq h0 b) (v n)))
let index #a b n =
  let s = !b.content in
  Seq.index s (v b.idx + v n)
```

We swap this F* model with a low-level implementation. 
`buffer int` becomes `int*` and `index b i` becomes `b[i]`. 
High-level verification for low-level code (3)

Our low-level, sequence-based buffer model.

\[
\text{val index} : \#a:\text{Type} \to b:\text{buffer} a \to n:\text{UInt32}.t \{ v \ n < \text{length} \ b \} \to \\
\text{Stack} a \\
\quad (\text{requires (fun} \ h \to \text{live} \ h \ b)) \\
\quad (\text{ensures (fun} h0 \ z \ h1 \to \text{live} \ h0 \ b \ \text{\&\&} \ h1 = h0 \\
\quad \text{\&\&} \ z = \text{Seq.index} (\text{as_seq} \ n ^2 \ b) \ (v \ n)))
\]

\[
\text{let index} \#a \ b \ n = \\
\quad \text{let} \ s = !b.\text{content} \text{ in} \\
\quad \text{Seq.index} s \ (v \ b.\text{idx} + \ v \ n)
\]

We swap this F* model with a low-level implementation. buffer \text{int} becomes int* and index \text{b i} becomes b[i].
The formalization of Low* to Clight
With a diagram

Disclaimer: these steps are supported by hand-written proofs.
Side-channel resistance
What are we protecting against

- We want to guard against some **memory** and **timing** side-channels
- Our **secret** data is at an **abstract** type
- By using **abstraction**, we can **control** what operations we allow on secret data
Abstraction to the rescue

Our module for secret integers exposes a handful of audited, carefully-crafted functions that we trust have secret-independent traces.

(* limbs only ghostly revealed as numbers *)

val v : limb -> Ghost nat

val eq_mask: x:limb -> y:limb ->
   Tot (z:limb{if v x <> v y then v z = 0 else v z = pow2 26 - 1})

By construction, the programmer cannot use a limb for branching or array accesses.
What we show

We model **trace events** as part of our reduction.

$$\ell ::= \cdot \mid \text{read}(b, n, f) \mid \text{write}(b, n, f) \mid \text{brT} \mid \text{brF} \mid \ell_1, \ell_2$$

*Note: this does not rule out ALL side channels!*
Secret-independence: an intuition

A type-indexed relation $v_1 \equiv_\tau v_2$ over values:

\[
\begin{align*}
n &\equiv_{\text{int}} n \\
v_1 &\equiv_{a} v_2 \\
&\ldots
\end{align*}
\]

**Intuition:** terms are related if they only differ on sub-terms at secret types.

**Main theorem:** functions, when applied to related values in related stores, have related reductions and emit the same traces.

*Note: this only goes up to CompCert Clight*
The KreMLin tool
A compiler from F* to readable C

The KreMLin facts:

- about 12,000 lines of OCaml
- carefully engineered to generate readable C code
- essential for integration into existing software.

Destroys modularity upon request for the sake of performance.

- Monomorphization
- Inlining
- Recombining modules (static inline)
- Recombining functions (intra-procedural optimizations)

So far, about 50k lines of C generated.
Evaluation
A word on HACL*

Our flagship crypto algorithms library. Available standalone, as an OpenSSL engine, or via the NaCl API.

- Implements Chacha20, Salsa20, Curve25519, X25519, Poly1305, SHA-2, HMAC
- 7000 lines of C code
- 23,000 lines of F* code
- Performance is comparable to existing C code (not ASM)
- Some bits are in the Firefox web browser!

Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, Benjamin Beurdouche
HACL*: A Verified Modern Cryptographic Library
CCS’17
A chart showing the performance of X25519 operations per second (ops/s) for different input bytes. The vertical axis represents the number of operations per second in the form of $1.5 \cdot 10^5$. The chart compares HACL* and OpenSSL with different input bytes.
input bytes

ChaCha20 1000s of bytes/s (higher is better)

HACL
OpenSSL
ChaCha20 1000s of bytes/s (higher is better)

- HACL
- HACL -vec
- OpenSSL
- OpenSSL ASM

input bytes

16  64  256  1024  8192  16384
A word on Vale

Vale: **Verified Assembly Language for Everest**

Some of the performance gap may be closed using intrinsics. But for CPU-specific instructions: use a dedicated language.

Barry Bond, Chris Hawblitzel, Manos Kapritsos, K. Rustan M. Leino, Jacob R. Lorch, Bryan Parno, Ashay Rane, Srinath Setty, Laure Thompson

Vale: Verifying High-Performance Cryptographic Assembly Code

USENIX’17
A word on the TLS record layer

We have declared victory on the TLS record layer. It uses HACL*.

Full cryptographic games and proofs.

Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cedric Fournet, Markulf Kohlweiss, Jianyang Pan, Jonathan Protzenko, Aseem Rastogi, Nikhil Swamy, Santiago Zanella-Beguelin, Jean-Karim Zinzindohoue. Implementing and Proving the TLS 1.3 Record Layer Oakland (S&P) 17
Future plans

- **HACL***
  - more algorithms (P-curves)
  - more integration (e.g. NSS)
- **miTLS**, our TLS library in F* (WIP)
  - currently available as an alternate SSL backend for curl or within Nginx
  - finish lowering the protocol layer into Low*
- low-level **parsers** (e.g. ASN.1) (WIP)
Your future plans

It’s all on GitHub!

• https://www.github.com/FStarLang/FStar
• https://www.github.com/FStarLang/kremlin
• https://www.github.com/mitls/mitls-fstar
• https://www.github.com/mitls/hacl-star
• https://www.github.com/project-everest/vale
Thanks. Questions?