# Verified low-level programming embedded in F*


Jonathan Protzenko — Microsoft Research
Jean-Karim Zinzindohoué — INRIA Paris
Aseem Rastogi — Microsoft Research
Tahina Ramananandro — Microsoft Research
Peng Wang — MIT CSAIL
Santiago Zanella-Béguelin — Microsoft Research
Antoine Delignat-Lavaud — Microsoft Research
Cătălin Hriţcu — INRIA Paris
Karthikeyan Bhargavan — INRIA Paris
Cédric Fournet — Microsoft Research
Nikhil Swamy — 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.

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

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.
```

ave bugs.

# 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 corrupti
issue is not consi
```

have bugs.

# [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?

...have bugs.

```
OpenSSL Security Advisory [10 Nov 2016]
========================================

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

Severity: High

TLS connections us
attack by corrupti
issue is not consi
```

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

**Hanno Boeck via RT** rt at
*Fri Mar 25 12:10:32 UTC*

- Previous message: [o when using "no-asm
- Next message: [opens
- **Messages sorted by:**

```
Attached is a sample code
Poly1305 functions of ope

These produce wrong resul
the other three also on
```

## [openssl-dev] [openssl.org #4439] poly1305-x86.pl produces incorrect output

**David Benjamin via RT** rt at openssl.org
*Thu Mar 17 21:22:26 UTC 2016*

- Previous message: [openssl-dev] [openssl-users] Removing some systems
- Next message: [openssl-dev] [openssl.org #4439] poly1305-x86.pl produces incorrect output
- **Messages sorted by:** [ date ] [ thread ] [ subject ] [ author ]

```
Hi folks,

You know the drill. See the attached poly1305_test2.c.

$ OPENSSL_ia32cap=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
recommend doing the same in your own test harness, to
```

# Specifiying, programming and verifying Poly1305

Spec.Poly1305.fst

```
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
```

```
-:**-  Spec.Poly1305.fst   All (2,0)     Git-master  (F★ company)
Auto-saving...done
```

Hacl.Impl.Poly1305_64.fst

File Edit Options Buffers Tools F★ Help

memory safety

math spec

```
[@"substitute"]
val poly1305_last_pass_:
  acc:felem →
  Stack unit
    (requires (λ h → live h acc ∧ bounds (as_seq h acc) p44 p44 p42))
    (ensures (λ h0 _ h1 → live h0 acc ∧ bounds (as_seq h0 acc) p44 p44 p42
      ∧ live h1 acc ∧ bounds (as_seq h1 acc) p44 p44 p42
      ∧ modifies_1 acc h0 h1
      ∧ as_seq h1 acc == Hacl.Spec.Poly1305_64.poly1305_last_pass_spec_ (as_seq h0 acc)))
```

code

```
[@"substitute"]
let poly1305_last_pass_ acc =
  let a0 = acc.(0ul) in
  let a1 = acc.(1ul) in
  let a2 = acc.(2ul) in
  let open Hacl.Bignum.Limb in
  let mask0 = gte_mask a0 Hacl.Spec.Poly1305_64.p44m5 in
  let mask1 = eq_mask a1 Hacl.Spec.Poly1305_64.p44m1 in
  let mask2 = eq_mask a2 Hacl.Spec.Poly1305_64.p42m1 in
  let mask = mask0 &^ mask1 &^ mask2 in
  UInt.logand_lemma_1 (v mask0); UInt.logand_lemma_1 (v mask1); UInt.logand_lemma_1 (v mask2);
  UInt.logand_lemma_2 (v mask0); UInt.logand_lemma_2 (v mask1); UInt.logand_lemma_2 (v mask2);
  UInt.logand_associative (v mask0) (v mask1) (v mask2);
  cut (v mask = UInt.ones 64 ⟹ (v a0 ≥ pow2 44 - 5 ∧ v a1 = pow2 44 - 1 ∧ v a2 = pow2 42 - 1));
  UInt.logand_lemma_1 (v Hacl.Spec.Poly1305_64.p44m5); UInt.logand_lemma_1 (v Hacl.Spec.Poly1305_64.p44m1);
  UInt.logand_lemma_1 (v Hacl.Spec.Poly1305_64.p42m1); UInt.logand_lemma_2 (v Hacl.Spec.Poly1305_64.p44m5);
  UInt.logand_lemma_2 (v Hacl.Spec.Poly1305_64.p44m1); UInt.logand_lemma_2 (v Hacl.Spec.Poly1305_64.p42m1);
  let a0' = a0 -^ (Hacl.Spec.Poly1305_64.p44m5 &^ mask) in
  let a1' = a1 -^ (Hacl.Spec.Poly1305_64.p44m1 &^ mask) in
  let a2' = a2 -^ (Hacl.Spec.Poly1305_64.p42m1 &^ mask) in
  upd_3 acc a0' a1' a2'
```

proof

-:**- **Hacl.Impl.Poly1305_64.fst** 55% L394 Git-master (F★ FlyC- company ElDoc Wrap)

Poly1305_64.c

File Edit Options Buffers Tools C Help

```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 )0xfffffffffff;
  uint64_t r0 = a0 >> (uint32_t )44;
  uint64_t a1_ = (a10 + r0) & (uint64_t )0xfffffffffff;
  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 i0 = acc[0];
  uint64_t i1 = acc[1];
  uint64_t i0_ = i0 & (((uint64_t )1 << (uint32_t )44) - (uint64_t )1);
  uint64_t i1_ = i1 + (i0 >> (uint32_t )44);
  acc[0] = i0_;
  acc[1] = i1_;
  uint64_t a00 = acc[0];
  uint64_t a1 = acc[1];
  uint64_t a2 = acc[2];
  uint64_t mask0 = FStar_UInt64_gte_mask(a00, (uint64_t )0xffffffffffb);
  uint64_t mask1 = FStar_UInt64_eq_mask(a1, (uint64_t )0xfffffffffff);
  uint64_t mask2 = FStar_UInt64_eq_mask(a2, (uint64_t )0x3ffffffffff);
  uint64_t mask = mask0 & mask1 & mask2;
  uint64_t a0_0 = a00 - ((uint64_t )0xffffffffffb & mask);
  uint64_t a1_0 = a1 - ((uint64_t )0xfffffffffff & mask);
  uint64_t a2_0 = a2 - ((uint64_t )0x3ffffffffff & mask);
  acc[0] = a0_0;
  acc[1] = a1_0;
  acc[2] = a2_0;
}
```

-:**- **Poly1305_64.c** 49% L272 Git-master (C/l company A

# 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. |
|---|

# 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) (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.

> preservation of the stack structure

```
effect Stack (a:T                                        post a))) =
  STATE a (fun (p
    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) (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)
  /\ (for                                  ==>
    Heap.                                 Map.sel m1.h r))
```

the tip remains the same

Preserves the layout of the stack and doesn't allocate in any caller frame.

# High-level verification for low-level code (3)

Our low-level, sequence-based buffer model.

```
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 buf

spatial
safety

```
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.

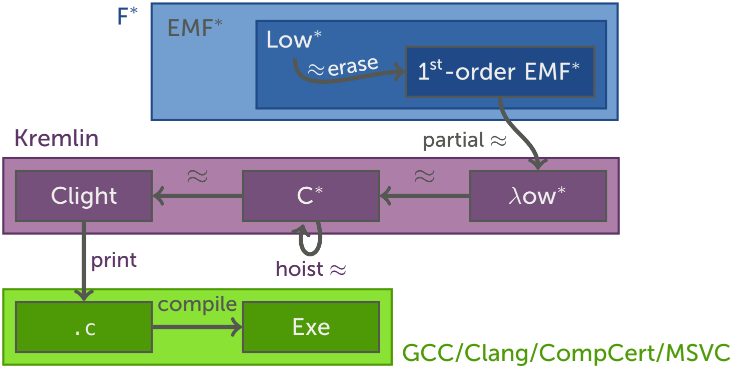
```
val index: #a:Type -> b:buffer a -> n:UInt32.t{v n < length b} ->
  Stack a
    (requires (fun h -> live h b))
    (ensures (fun h0 z h1 -> live h0 b /\ h1 == h0
      /\ z == Seq.index (as_seq h0 b) (v n)))
let index #a b n =
  let s = !b.content in
  Seq.index s (v b.idx + v n)
```

temporal safety

We swap this F* model with a low-level implementation.
**`buffer int`** becomes **`int*`** and **`index 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.

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

*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:

$$n \equiv_{\mathsf{int}} n$$
$$v_1 \equiv_a v_2$$
$$\ldots$$

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.

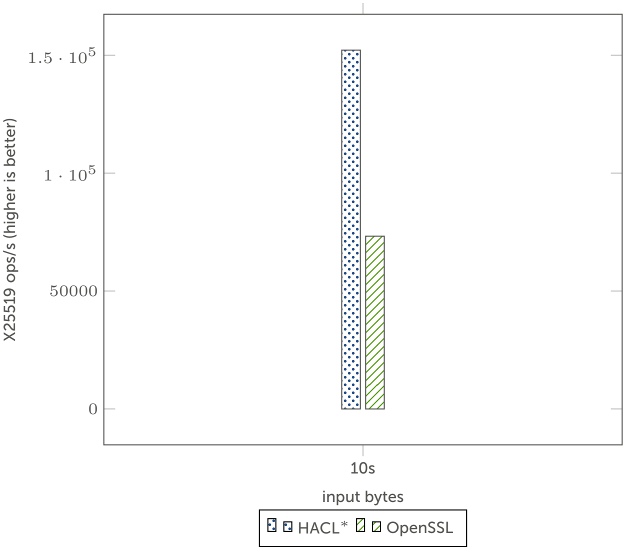
# Evaluation

# A word on HACL*

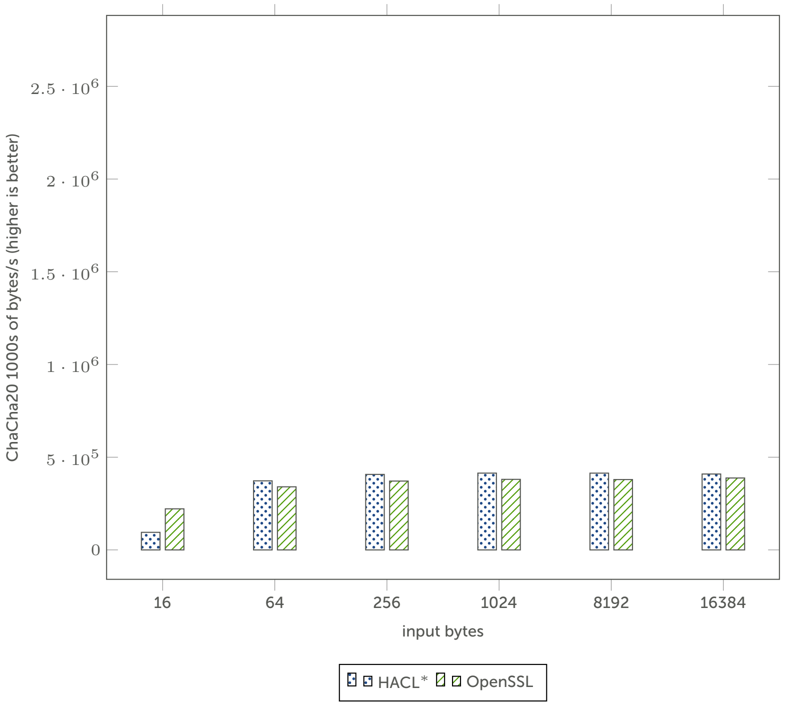
Our flagship crypto algorithms library. Available standalone, as an OpenSSL engine, or via the NaCl API.

- 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!

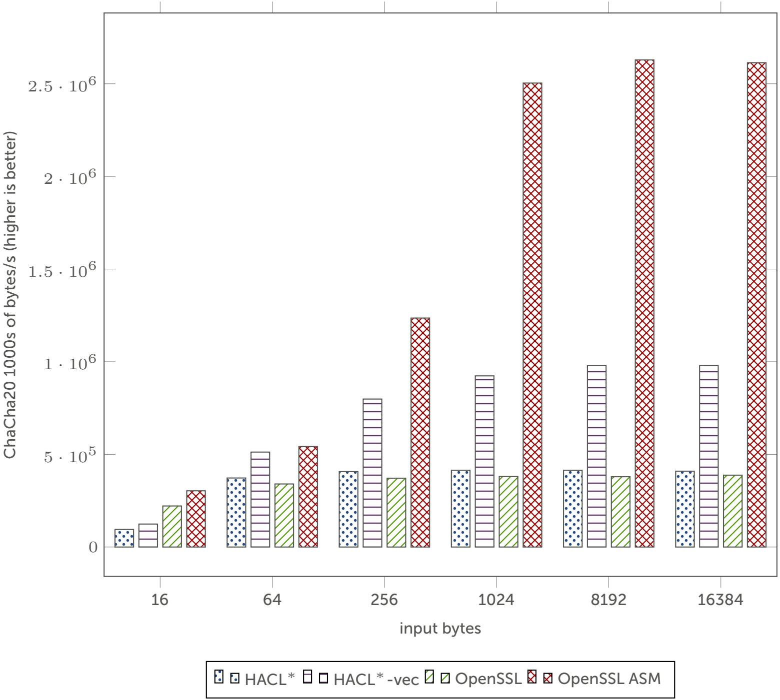
Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, Benjamin Beurdouche
HACL*: A Verified Modern Cryptographic Library
CCS'17



X25519 ops/s (higher is better)
$1.5 \cdot 10^5$
$1 \cdot 10^5$
50000
0
10s
input bytes
HACL*
OpenSSL



ChaCha20 1000s of bytes/s (higher is better)
0
$5 \cdot 10^5$
$1 \cdot 10^6$
$1.5 \cdot 10^6$
$2 \cdot 10^6$
$2.5 \cdot 10^6$
16
64
256
1024
8192
16384
input bytes
HACL*
OpenSSL


ChaCha20 1000s of bytes/s (higher is better)
$2.5 \cdot 10^6$
$2 \cdot 10^6$
$1.5 \cdot 10^6$
$1 \cdot 10^6$
$5 \cdot 10^5$
0
16
64
256
1024
8192
16384
input bytes
HACL*
HACL* -vec
OpenSSL
OpenSSL ASM

# 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.

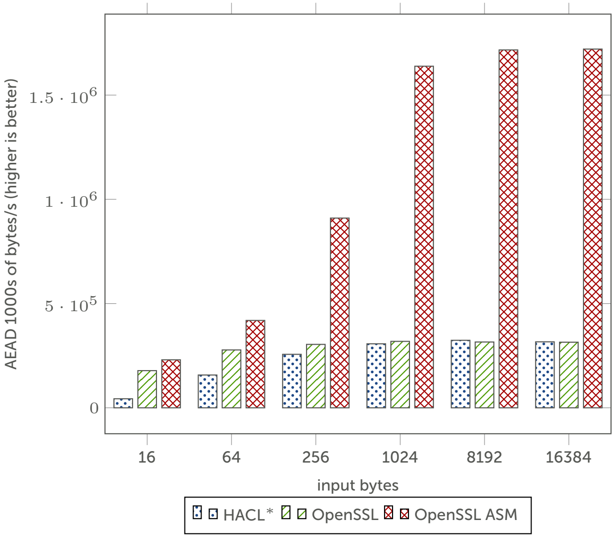
Barry Bond, Chris Hawblitzel, Manos Kapritsos, K. Rustan M. Leino, Jacob R. Lorch, Bryan Parno, Ashay Rane, Srinath Setty, Laure Thompson
Vale: Verifying High-Performance Cryptographic Assembly Code
USENIX'17

# 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



AEAD 1000s of bytes/s (higher is better)
$1.5 \cdot 10^6$
$1 \cdot 10^6$
$5 \cdot 10^5$
0
16
64
256
1024
8192
16384
input bytes
HACL*
OpenSSL
OpenSSL ASM

# 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?