Microsoft Research

Each year Microsoft Research hosts hundreds of influential speakers from around the world including leading scientists, renowned experts in technology, book authors, and leading academics, and makes videos of these lectures freely available. 2016 © Microsoft Corporation. All rights reserved.
Project Everest

theory meets reality

Jonathan Protzenko
Microsoft Research

Project Everest
INRIA Paris
MSR Redmond, Cambridge, Bangalore
CMU

May 14th, 2018
Everest:
Deploying Verified-Secure Implementations in the HTTPS Ecosystem
CIAO
TRATTORIA - PIZZERIA
WINE - BAR

EVEREST
Scaffolding Inc.
1150 LONGWOOD AVE. BRONX, NY 10474
(718) 328-1004

Ciao Vineria Con Cucina
1. Verification challenges in the HTTPS ecosystem
2. A formalized toolchain for delivering C and ASM code
3. Tooling support for programmer productivity
4. Stories from the “real world”
Challenges in the HTTPS ecosystem
What is there to verify?

Everest: Expedition for a VERified Secure Transport

- A verified secure HTTPS ecosystem: this is huge
- Just focusing on TLS and QUIC and their dependencies
- But first, some background on TLS
HTTPS: the TLS protocol

TLS stands for *transport layer security*.

Two different kinds of beasts:
- the protocol layer
- the record layer
HTTPS: the QUIC protocol

Based on **UDP** instead of **TCP**.

Two goals: latency and multiplexing.

Re-uses the **handshake** from TLS 1.3 (0RTT) but then does its own thing for the stream data.
Why is it hard? (The crypto)

Implement efficient arithmetic over large numbers (bignums).

- Optimized bitwise operations
- Each bignum has its own optimized representation (reuse)
- Difficult to exhaustively test

Goal: functional correctness (implies memory safety) + side-channel resistance.
Why is it hard? (Poly1305 example)

These heavily optimized C implementations have bugs.
Why is it hard? (Poly1305 example)

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

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

Severity: High

TLS connections using 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-users] Removing some systems
- Next message: [openssl-dev] [openssl.org #4489] poly1305-x86.pl produces incorrect output
- Messages sorted by: [date] [thread] [subject] [author]

Attached is a sample code of OpenSSL's poly1305 functions of openssl
These produce wrong results, the other three also on Windows.

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

$ OPENSSL_LA2cap=0 /poly1305_test2
PASS
$ ./poly1305_test2
Poly1305 test failed.
got: 2637408f63086ea73f971e3425e820
expected: 2637408f63086ea73f971e3425e820

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
recording. Once that is done, I think we can continue our process.
Why is it hard (record layer)

Provide a safe cryptographic **functionality** by combining primitive blocks. Example: AEAD.

- **Multiplex** between different algorithms (AES-GCM, Chacha-Poly).
- Safely **combine** the cryptographic primitives.
- Reason about integrity, authenticity, confidentiality.

**Goal:** cryptographic strength + side-channel resistance.
Why is it hard (the handshake)

Provide a correct **state machine** that manages keys properly.

- Need for **speed**: 0-RTT and 0.5-RTT
- Multiple ways to **derive** keys (PSK, forward secrecy, rekeying)
- Handle **choice** of algorithms, **versions** (1.2 vs. 1.3)

**Goal:** cryptographic **security**.
Why is it hard (the handshake)

Parse messages following the RFC.

- Parsers are notoriously error-prone.
- Need to interop, but hard to exhaust all the code-paths.
- RFC informal.

Goal: memory safety ("if it interops, it interops").
Why is it hard (QUIC)

Implement retransmission, windows, error correction, out-of-order frames, etc.

- Low-level systems programming
- Data structures: “inline” doubly-linked lists with ugly C macros
- Concurrency with different streams
- Interaction with the rest of the OS
- Risk of integer overflow

Goal: memory safety
In short...

Many **different types** of guarantees. The HTTPS ecosystem really is a minefield.

**Status:**
- crypto: **verified** (some algorithms)
- record layer: **verified**
- handshake: **in progress**
- parser: **nearing completion**
- QUIC: **scheduled**
- PKI: **on the horizon**
A formalized toolchain
With a diagram

Disclaimer: these steps are supported by hand-written proofs.
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.
A sample cryptographic operation: Poly1305

Poly1305 is a message authentication code.

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

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.

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

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 \( (a \times 2^{130} + b) \mod p = (a + 4a + b) \mod p \) for reductions
- unfolds the loop
Specifying, programming and verifying Poly1305

```
module Spec.Poly1305
let prime = pow2 130 - 5
let 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)) \text{ \textquote{fadd}} \text{little-endian } w
let rec poly (txt:txt) (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 \text{\textquote{fadd}} a) \text{ fmul } r
```

J. Protzenko et al. — MSR
Project Everest: theory meets reality
May 14th, 2018 22 / 59
Insights about our formalization
Our low-level, stack-based memory model.

```haskell
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 a1.
      (pre h /
        post h a a1 /
        equal_domains h h1) === p a a1))

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:T) = STATE a (fun (p:post a) \ pre h \ (\forall 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.
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 \ (forall 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
  \(\forall r.\text{Set.equiv}(\text{Map.domain} m0.\text{h})(\text{Map.domain} m1.\text{h})\)
  \(\forall r.\text{Heap.equiv}(\text{Map.sel m0.\text{h} r})(\text{Map.sel m1.\text{h} r})\)
```

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.

```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].
High-level verification for low-level code (3)

Our low-level, sequence-based buffer safety

```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**.

```scala
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 h 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].
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.

\[ \ell ::= \cdot \mid \text{read}(b, n, \bar{f}) \mid \text{write}(b, n, \bar{f}) \mid \text{br}T \mid \text{br}F \mid \ell_1, \ell_2 \]

Note: this does not rule out ALL side channels!
The KreMLin tool
A compiler from F* to readable C

The KreMLin facts:
- about 14,000 lines of OCaml
- carefully engineered to generate readable C code
- essential for integration into existing software.

Design:
- relies on the same Letouzey-style erasure from F*
- one internal AST with several compilation passes
- abstract C grammar + pretty-printer
- small amounts of hand-written C code (host functions)

So far, about 120k lines of C generated.
Evaluation
A word on HACL*

Our 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
Vale: extensible, assembly language verification
Crypto performance: OpenSSL vs. Vale

- AES: OpenSSL with SIMD, AES-NI
- Poly1305 and SHA-256: OpenSSL non-SIMD assembly language
  - Same assembly code for OpenSSL, Vale

![Graphs showing throughput comparison between OpenSSL and Vale for AES, Poly1305 MAC, and SHA-256 hash.](image-url)
③ Tooling support
Cryptography: a (too) good example

- Crystal clear math spec
- Trivial allocation patterns
- The code is naturally low-level

A driver that informed the design and implementation of Low*.
But! ...beyond cryptography

- Allocation patterns are **more complex**
- The code is naturally **higher-level**
- Surprise: people actually **do not want** to write C in F*
- **Strong push** for more tooling support
A point in the design space

- **Reality**: moving beyond the paper formalization
- **Tension**: the tooling is not verified
- **Claim**: priority ordering: high-risk source, lower risk tooling

**Productivity/scaling vs. Verified toolchain**
Tooling support: killing abstraction

Abstraction = good for verification
No Abstraction = good for compilation

- At the module level (-bundle)
- At the function level (inline_for_extraction)

This triggers enough compiler optimizations to fulfill the original promise.
Tooling support: data types

Or: “programmer productivity”.

- Tuples, inductives (tagged unions) are supported
- Four (!) different compilation schemes
- Use at your own risk (MSVC! CompCert! x86 ABI!)
- Requires:
  - monomorphization
  - implementation in KreMLin of recursive equality predicates
  - mutual recursion; forward declarations
Tooling support: misc

- Type abbreviations
- C loops (syntactic closures for bodies)
- Removal of uu
- Optimal visibility
- Removal of unused function and data types arguments
- Passing structures by reference
Tooling support: conclusion

so... none of this is rocket science
but... it’s a slippery slope
idea have a mode that disables cosmetic optimizations
to do differential testing.

There is a constant tension (e.g. tail-rec).

There is hope: all the bugs found so far were either in the formalization, in unverified, glue code, or in the compiler.
Four: Two stories about the real world

1. Firefox
2. Windows Kernel mode
Firefox (1): the code

- These people **actually** read our code
- Stringent **coding standards**
  - parentheses
  - unused variables
  - unused parameters
- Cosmetic (**indentation**, no clang-format)
- More fundamentally: **no recursion** and **no uint128 support** (cross-platform)
- Still need to implement **const** support ($\neq$ our formalization)
Firefox (2): the infrastructure

- They used a Docker VM to put the toolchain under CI
- No one can modify the code directly
- One student at INRIA supports them
- Minimize the hand-written glue code (FStar.h and Kremlinlib.h)
Kernel mode (1): why?

- Lower latency (usermode/kernelmode transitions) + connection management in-kernel
- Pooling of connections to the same domain
- Better security (keys in OS memory)
- Primitive IO API support
- Makes it available to other drivers
The first problem was the **Microsoft Compiler (MSVC)**

- VS2017 has decent **C11** support
- No **uint128** type
- No **variable-length arrays**
- Arbitrary nested **struct depth**
- Unpredictable **tail-calls and struct passing optimizations**
Kernel mode (3): all the other things

A lot of things were not captured by our formalization.

- excessive stack consumption: limit is 12k in kernel mode (value structs, lack of tail-calls)
- abuse of recursion: byte-by-byte copy is great for verification but...
- need to offer C-like APIs: some amount of glue code

Stack overflows are not good...
Kernel mode (4): misc

- No C runtime means different APIs
- Logging APIs
- Symbol collisions
- MSVC compiler bug
- C standard library bug
5 Conclusion
In hindsight

- The paper is only half the work
- Prioritize verification effort
- Nothing beats good CI and testing
- Tooling matters
Your future plans

It's all on GitHub!

- https://www.github.com/FStarLang/FStar
- https://www.github.com/project-everest/vale
- 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/everest
Thanks.
Questions?