Lecture 4 notes

Butler Lampson
6.826
September 10, 2020

The paper is
Everest: Towards a Verified, Drop-in Replacement of HTTPS, 2nd Summit on Advances in Programming Languages (SNAPL), Asilomar, May 2017

The problem: current network security code (SSL/TLS) is insecure

SSL/TLS uses on the Internet:
- Browser-server security
- Most other connections

54 TLS vulnerabilities in a typical year

Major flaws: Heartbleed, FREAK, Logjam
- Heartbleed: buffer overrun from mishandled length field in added message
- FREAK: allow old, weak crypto for compatibility
- Logjam: precompute tables to make it faster to find discrete logs

How serious is it?

Hunters and bear
Threat model: MiM makes it much harder; weak defaults make it harder
Worst case: just send a bad message. Now everyone is vulnerable

Contrast # vulnerable with press

Intelligence: ideal is invisible, permanent leak, even if slow
Finance: can lose billions, from bitcoin hacks or compromised credentials (Bangladesh)
Shopping

Everest

Client: Browser Server: Amazon

HTTPS

TLS

OS

Hardware: CPU *NIC*

. *Network*

TCB: Everything except the network and NIC

Later we'll talk about how to get the OS out of the TCB with enclaves

Why is this interesting? You have less control in network

Everest: make HTTPS/TLS secure

Still depend on OS, hardware

Depend on crypto algorithms

Search for "Everest project" to find the Microsoft Research and Github pages.
Everest today: many parts

- F*, a verification language for effectful programs
- miTLS, reference implementation of the TLS protocol in F*
- KreMLin, a compiler from a subset of F* to C
- HACL*, a verified library of cryptographic primitives written in F*
- Vale, a domain-specific language for verified cryptographic primitives in assembly
- EverCrypt, a verified crypto provider with an agile, multi-platform, self-configuring cryptographic API.
- EverParse, automatically generates verified parsers and serializers for binary data formats

The hope: definitively solve this problem

Challenges:

- Correctness
- Deployment

Approach:

- Write specs, code in **F***: Coq – tactics + SMT solver (now adding tactics)
  - Code in Low* ⊆ F*: Erase fancy stuff, extract to C, compile with …
- Proof in F*
- **Vale** for verified assembly code

Could it work? In real life? Against academic attacks?

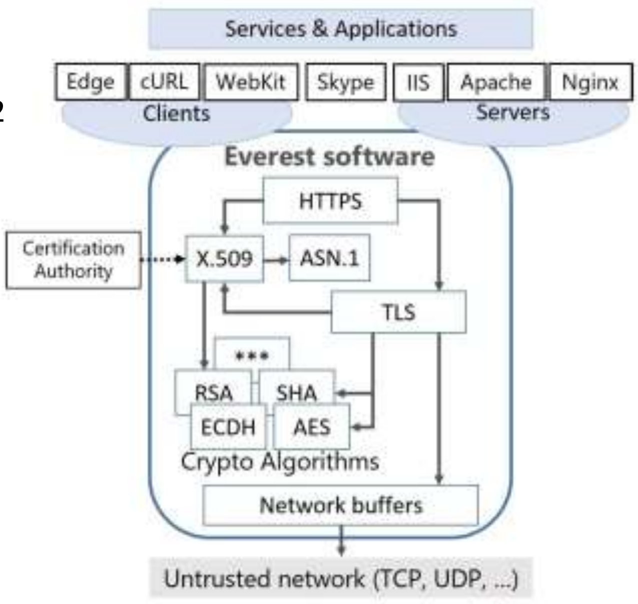
Fisher K, Launchbury J, Richards R. 2017
The HACMS program: using formal methods to eliminate exploitable bugs.
Phil. Trans. R. Soc. A 375, 2017

Correctness for TLS

Spec: $socket\ =\ connect(to\text{:string}, using\text{:cryptoParameters})$
ha, ha: Georgiev et al. The most dangerous code in the world, 2012

Proof: In F*

Dependencies: What to worry about after F* says PROVED
    spec, F*, certs, CPU semantics, OS



Dangerous code

Don’t hack code to disable validation for testing
Don’t take defaults; specify what you want
Test aggressively—certificates are complicated

Crypto

Primitives

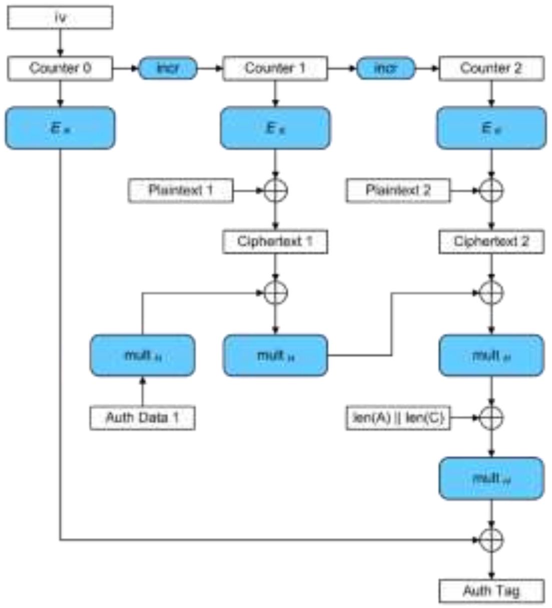
$dec_K\left(enc_K(m)\right) = m$ and dec(anything else) $= error$

$verify_K\left(sign_{K^{-1}}(m)\right) = true$ and $verify$(anything else) $= error$

This is ideal crypto. Real crypto delivers this with
some probability $1 - \epsilon$, and
except for one-time pads, some risk that hard problems get solved
such as factoring, discrete log, or multi-round jumbling

Build this for arbitrary messages from basic algorithms
These do one block or add one block to a MAC
These have mathematical specs

Performance is important
State of the art for data: .8 cycles/byte
Achieved by best OpenSSL code, and by Evercrypt



Side channels

Because the spec is not complete.
    Abstractions don’t keep secrets; they leak
    If you worry about side channels, don’t share resources

Timing dependent: E avoids this
    Speculative

Radiation: EM, acoustic
Power
…

Crypto protocol

Key exchange: DH gives you a shared secret K_{DH}, but you don't know with whom

Authentication: keys speak for principals, $K \Rightarrow P$

- cert **says** $K \Rightarrow string$ (hostname) e.g., amazon.com
  - cert is $sign_{K_{\text{verisign}}}(K_{az} \Rightarrow \text{amazon.com})$
- $verify_K(K_{DH} \Rightarrow K \text{ at time } t)$
- Certs are complicated: many features, complex encoding
- Cert chains are even more complicated

Many options for

- Legacy
- Performance
- "Convenience"

Parsing

- Want parser and serializer like crypto: $p\big(s(m)\big) = m$ and $p(\text{anything else}) = error$
- With complex formats, lots of chances for bugs
  - Especially when you are going to sign (a hash of) s.
  - You have K **says** $s(m)$ and you want $host$ **says** m

Everparse: Verified parsers for binary formats

TLS

Handshake: set up and authenticate keys
Record protocol: transmit data

Handshake idea

Negotiate crypto parameters: Algorithm, key lengths, …
Use Diffie-Hellman to get a shared master key K_{master}
Authenticate a host identity key: $K_{\text{host}} \Rightarrow hostname$
Using certificates
Authenticate the whole: $sign_{K_{\text{host}}}(K_{\text{master}}, crypto\ parameters)$

Many complications for legacy, efficiency, rekeying, …

Record protocol

Encrypt record with K, send it through network, decrypt and authenticate with K.

HTTPS

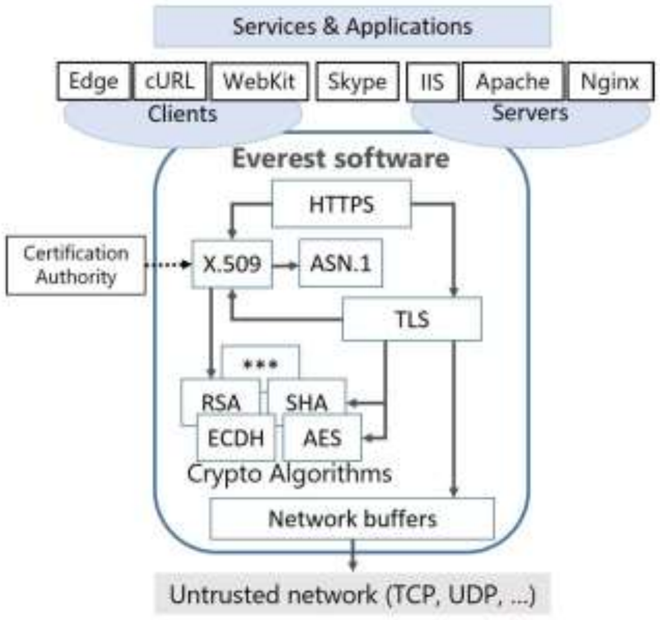
Delivers good pages to browser client.

Need to call TLS correctly

Notes

Game-playing proofs of security
TLS 1.3 record layer in Low*
F*, a dependently typed language for programming, meta-programming, and proving at a high level
Low* -> C via Kremlin
Vale for verified assembly
Performance and, sometimes, side channel resistance
We compile all our code to source-like C and assembly

After verification, in support of incremental deployment, our code is extracted by verified tools to C and assembly, and compiled further by off-the-shelf C compilers (e.g., gcc and clang, but also, at a performance cost, verified compilers like CompCert [18]) and composed with adapters that interface our verified code with existing software components, like the web browsers, servers and other communication software shown in the Figure 1



Details

From the Github header

Project Everest is the combination of the following projects. Read below for an easy way to install all these projects together.

- F*, a verification language for effectful programs
- miTLS, reference implementation of the TLS protocol in F*
- KreMLin, a compiler from a subset of F* to C
- HACL*, a verified library of cryptographic primitives written in F*
- Vale, a domain-specific language for verified cryptographic primitives in assembly
- EverCrypt, a verified crypto provider that combines HACL* and Vale via an agile, multi-platform, self-configuring cryptographic API.
- EverParse, a library and tool to automatically generate verified parsers and serializers for binary data formats

When combined together, the projects above generate a mixture of C and assembly code that implements TLS 1.3, with proofs of safety, correctness, security and various forms of side-channel resistance.

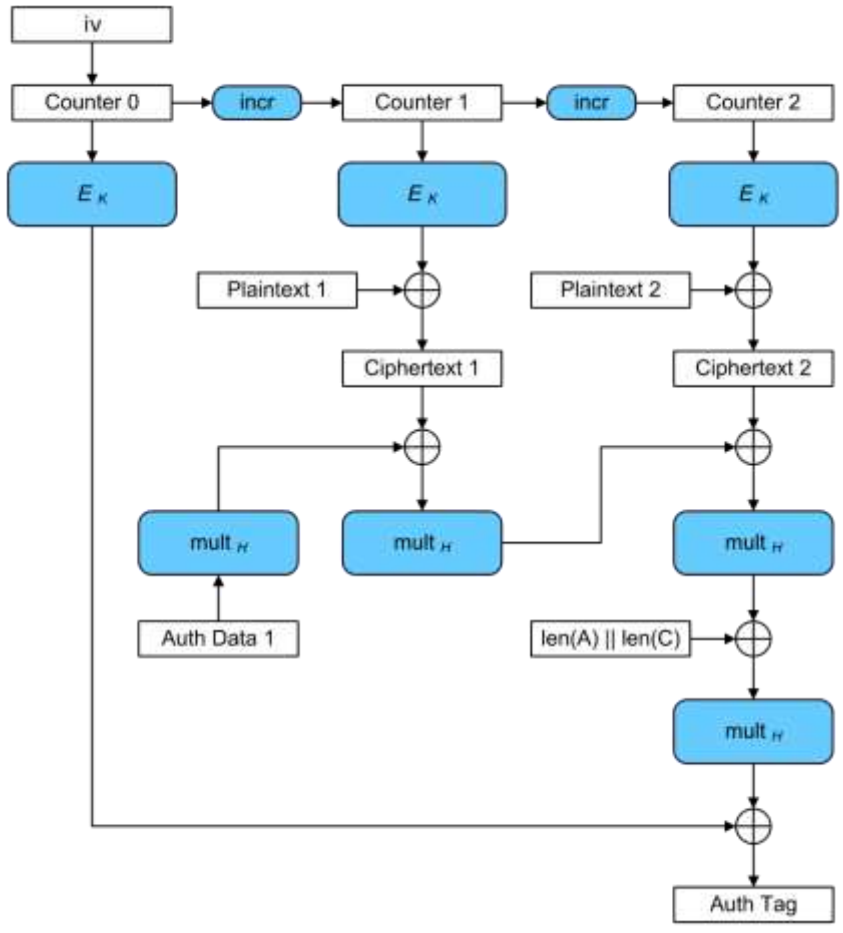
Everest is a work in progress. We generate C and assembly code for TLS-1.3, but the verification is not complete.

- The TLS 1.3 **handshake** verification is a work in progress
- We have completed verification of the TLS 1.3 **record layer**; it currently extracts to C.
- Several cryptographic **assembly routines**, including AES-GCM, Poly1305, AES and SHA2, are verified and extract to assembly via Vale. (**USENIX-17**, **POPL-19**)
- **HACL*** provides verified C code for multiple other primitives such as Curve25519, Chacha20, Poly1305 or HMAC.
  - Everest code is deployed in several contexts.
  - Code from the HACL* crypto library and EverCrypt crypto provider is deployed in Mozilla Firefox, the Wireguard VPN, the upcoming Zinc crypto library for the Linux kernel, the MirageOS unikernel, and in the Tezos and Concordium blockchains.
  - The miTLS protocol stack powers Microsoft's primary implementation of the QUIC transport protocol.

Crypto

https://en.wikipedia.org/wiki/Galois/Counter_Mode

Blocks are numbered sequentially, and then this block number is combined with an initialization vector (IV) and encrypted with a block cipher E, usually AES. The result of this encryption is then XORed with the plaintext to produce the ciphertext. The ciphertext blocks are considered coefficients of a polynomial which is then evaluated at a key-dependent point H, using finite field arithmetic. The result is then encrypted, producing an authentication tag that can be used to verify the integrity of the data. The encrypted text then contains the IV, ciphertext, and authentication tag.



Impressive performance results are published for GCM on a number of platforms. Käsper and Schwabe described a "Faster and Timing-Attack Resistant AES-GCM"[13] that achieves 10.68 cycles per byte AES-GCM authenticated encryption on 64-bit Intel processors. Dai et al. report 3.5 cycles per byte for the same algorithm when using Intel's AES-NI and PCLMULQDQ instructions. Shay Gueron and Vlad Krasnov achieved 2.47 cycles per byte on the 3rd generation Intel processors. Appropriate patches were prepared for the OpenSSL and NSS libraries.[14]

Vale paper:
Altogether, counting comments and white space, our verified AES-GCM implementation requires 339 lines of specification, 2020 lines of proof libraries, 73 Vale procedures, over 1100 lines of Low⋆ code, and more than 4400 lines of Vale code.

	AES-GCM-128	AES-GCM-256
OpenSSL (SIMD, AESNI/PCLMULQDQ)	6414	4730
Vale/F⋆ (SIMD, AESNI/PCLMULQDQ)	991	935

But the Evercrypt paper shows almost equal performance for the “targeted” AEAD at about .8 cycles/byte. This is not the limiting factor for performance of TLS or QUIC.

Parsing

EverParse yields efficient zero-copy implementations, usable both in F* and in C.We evaluate it in practice by fully implementing the message formats of the Transport Layer Security standard and its extensions (TLS 1.0–1.3, 293 datatypes) and by integrating them into MITLS, an F* implementation of TLS. We illustrate its generality by implementing the Bitcoin block and transaction formats, and the ASN.1 DER payload of PKCS #1 RSA signatures.

Because they are directly exposed to adversarial inputs, parsers are often among the most vulnerable components of security applications. … When parsing is on the critical path of an application’s performance … developers may be forced to write and maintain their own parsers and serializers in low-level unsafe languages like C.
Nonmalleable: valid messages have unique representations.

Cryptographic mechanisms provide (serialized) byte string authentication, whereas applications rely on (parsed) message authentication. Hence, correctness and runtime safety are not sufficient to preserve authentication: a correct parser may accept inputs outside the range of the serializer, or multiple serializations of the same message, which may lead to subtle, and sometimes devastating, vulnerabilities.
A parser is *correct* with respect to a serializer when it yields back any formatted message: :$\forall m \in \mathcal{V} \mid p\big(s(m)\big) = m$, and *exact* when it accepts only serialized messages: $p^{-1}(\mathcal{V}) = s(\mathcal{V})$. Parsers may also be considered on their own. A parser is non-malleable (or injective) when it accepts at most one binary representation of each message: $\forall x, y \in \{0,1\}^* \mid p(x) = p(y) \Rightarrow (x = y \vee p(x) = \perp)$, and complete (or surjective) when it accepts at least one binary representation of each message:$p(\{0,1\}^*) \setminus \{\perp\} = \mathcal{V}$. If p is a non-malleable parser for V , then $p^{-1}$is a serializer over $p(\{0,1\}^*) \setminus \{\perp\}$.

Heartbleed (which is estimated to have affected up to 55% of the top internet websites [17]) is a simple buffer overrun caused by improper validation of the length field in the TLS messages defined in OpenSSL’s implementation of the heartbeat protocol extension.

PKCS #1 v1.5 defines a standard for hashing and padding the message to sign: given an arbitrary message m, it is first hashed into a digest h, then stored together with the identifier a of the hash algorithm. The signature is $\sigma = RSA((a, h) + padding)$. There’s padding, and if the parser doesn’t check that it’s right, there are lots of (a, h) such that

*F**

We aim for a language that spans the capabilities of interactive proof assistants like Coq, general-purpose programming languages like OCaml and Haskell, and SMT-backed semiautomated program verification tools like Dafny. This language would provide the nearly arbitrary expressive power of a logic like Coq's, but with a richer, effectful dynamic semantics. It would provide the flexibility to mix SMT-based automation with interactive proofs when the SMT solver times out.

Scripting proofs using tactics and metaprogramming: properties of pure programs are specified in expressive higher-order (and often dependently typed) logics, and proofs are conducted using various imperative programming languages.
Along a different axis, program verifiers like Dafny target both pure and effectful programs. They work primarily by computing verification conditions (VCs) from programs, usually relying on annotations such as pre- and post-conditions, and encoding them to automated theorem provers (ATPs) such as satisfiability modulo theories (SMT) solvers.
Meta-F* is a framework that allows F* users to manipulate VCs using *tactics*. More generally, it supports *metaprogramming*, allowing programmers to script the construction of programs, by manipulating their syntax and customizing the way they are type-checked.

Refinement types: $x : t\{\phi\}$ is the type of all x of type t such that $(\lambda x \mid \phi)(x)$.

One verification method that has eluded F* until now is separation logic, the main reason being that the pervasive "frame rule" requires instantiating existentially quantified heap variables, which is a challenge for SMT solvers, and simply too tedious for users.