Security Verification and cryptographic modelling in F*

Antoine Delignat-Lavaud
Catalin Hritcu
Danel Ahman
Everest*: Verified Drop-in Replacements for TLS/HTTPS
The HTTPS Ecosystem is critical

- Default protocol—trillions of connections
- Most of Internet traffic (+40%/year)
- Web, cloud, email, VoIP, 802.1x, VPNs, IoT...
The HTTPS Ecosystem is complex
The HTTPS Ecosystem is broken

- **20 years of attacks & fixes**
  - Buffer overflows
  - Incorrect state machines
  - Lax certificate parsing
  - Weak or poorly implemented crypto
  - Side channels
  - Implicit security goals
  - Dangerous APIs
  - Flawed standards

- **Mainstream implementations**
  - OpenSSL, SChannel, NSS, ...
  - Monthly security patches

---

Diagram: Vimeo.com video 804767257, CC BY-SA 4.0

- Services & Applications:
  - Edge, cURL, WebKit, Skype, IIS, Apache, Nginx

- Clients

- Untrusted network (TCP, UDP, …)

- Network buffers

- Crypto Algorithms:
  - RSA, SHA, ECDH, 4Q

- Certification Authority

- OSI Reference Model
A Timeline of Recent TLS Attacks

Crypto failures
- MD5

Protocol weaknesses
- Renegotiation Attack
- BEAST (Rogaway 02)
- CRIME
- ECDHE Cross-protocol Attack
- Lucky13
- POODLE
- EarlyCCS
- Heartbleed
- SLOTH
- Triple Handshake
- Logjam
- FREAK
- SKIP
- DROWN

Implementation bugs
- OpenSSL entropy
- RC4
- MD5
- RSA 512 bit
- SHA1
- ECDHE Cross-protocol Attack (Rogaway 02)
- BEAST
- CRIME
- Lucky13
- POODLE
- EarlyCCS
- Heartbleed
- SLOTH
- Triple Handshake
- Logjam
- FREAK
- SKIP
- DROWN

Years:
- 2007
- 2008
- 2009
- 2010
- 2011
- 2012
- 2013
- 2014
A Timeline of Recent PKI Failures

Crypto failures
- Debian OpenSSL entropy bug
- Bleichenbacher’s e=3 attack on PKCS#1 signatures

HashClash rogue CA (MD5 collision) Stevens et al.
- Flame malware NSA/GCHQ attack against Windows CA

512 bit Korean School CAs
- BERSerk (MSR—Inria)

Basic constraints not enforced (recurring catastrophic bug)

Name constraints failures
- OpenSSL null prefix

Usage-unrestricted VeriSign certificates
- DROWN KeyUsage

Formatting & semantics
- GnuTLS X509v1

CA failures
- VeriSign hack
- VeriSign NetDiscovery
- Comodo hack
- DigiNotar hack
- Trustwave
- TÜRKTRUST
- ANSSI

The SHApepning
- OpenSSL CVE-2015-1793

2006 2007 2008 2009 2010 2011 2012 2013
## Side Channel Challenge (Attacks)

<table>
<thead>
<tr>
<th>Protocol-level side channels</th>
<th>Traffic analysis</th>
<th>Timing attacks against cryptographic primitives</th>
<th>Memory &amp; Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS messages may reveal information about the internal protocol state or the application data</td>
<td>Combined analysis of the time and length distributions of packets leaks information about the application</td>
<td>A remote attacker may learn information about crypto secrets by timing execution time for various inputs</td>
<td>Memory access patterns may expose secrets, in particular because caching may expose sensitive data (e.g. by timing)</td>
</tr>
<tr>
<td>• Hello message contents (e.g. time in nonces, SNI)</td>
<td>• CRIME/BREACH (adaptive chosen plaintext attack) • User tracking • Auto-complete input theft</td>
<td>• Bleichenbacher attacks against PKCS#1 decryption and signatures • Timing attacks against RC4 (Lucky 13)</td>
<td>• OpenSSL key recovery in virtual machines • Cache timing attacks against AES</td>
</tr>
<tr>
<td>• Alerts (e.g. decryption vs. padding alerts) • Record headers</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Remote timing attacks are practical

2000 ... 2006 2007 2008 2009 2010 2011 2012 2013 2014

- Bleichenbacher
- Vaudenay
- AES cache timing
- Side-channel leaks in Web applications
- ECDSA timing
- Tag size
- CRIME
- Lucky13
- BREACH
- DROWN
Verified Components for the HTTPS Ecosystem

- Strong verified safety & security
- Trustworthy, usable tools
- Widespread deployment
Team Everest

Systems and Engineering

- Barry Bond
- Chris Hawblitzel
- Antoine Delignat-Lavaud
- Karthik Bhargavan
- Cédric Fournet
- Nik Swamy
- Santiago Zanella-Beguelin
- Benjamin Beurdouche
- Jean Karim Zinzindohoué
- Nadim Kobeissi

Security

- Bryan Parno
- Aseem Rastogi
- Catalin Hritcu
- Jonathan Protzenko
- Christoph Wintersteiger
- Leonardo de Moura
- Patrice Godefroid
- Danel Ahman
- Victor Dumitrescu
- Kenji Maillard
- Tahina Ramanandro

Cryptology

- Markulf Kohlweiss

Programming & Verification

- Santiago Zanella-Beguelin
- Antoine Delignat-Lavaud
- Cédric Fournet

Locations:
- Cambridge
- Bangalore
- Redmond
- Paris (INRIA)
- Pittsburgh (CMU)
TLS/HTTPS: Just a Secure Channel?

Crypto provable security (core model)
- One security property at a time—simple definitions vs composition
- Intuitive informal proofs
- Omitting most protocol details
  New models & assumptions required

RFCs (informal specs)
- Focus on wire format, flexibility, and interoperability
  Security is considered, not specified

Software safety & security (implementation)
- Focus on performance, error handling, operational security
  Security vulnerabilities & patches

Application security (interface)
- Lower-level, underspecified, implementation-specific. Poorly understood by most users.
  Weak configurations, policies, and deployments
High-Performance Verified Implementations

By implementing standardized components and proving them secure, we validate both their design and our code.

source code, specs, security definitions, crypto games & constructions, proofs...

verify all properties (using automated provers) then erase all proofs

extract low-level code, with good performance & (some) side-channel protection

interop with rest of TLS/HTTPS ecosystem
Traditionally, software is produced in this way: write some code, maybe do some code review, run unit-tests, and then hope it is correct. Hard experience shows that it is very hard for programmers to write bug-free software. These bugs are sometimes caught in manual testing, but many bugs still are exposed to users, and then must be fixed in patches or subsequent versions. This works for most software, but it’s not a great way to write cryptographic software; users expect and deserve assurances that the code providing security and privacy is well written and bug free.
We integrate miTLS & its verified crypto with Internet Explorer.

We run TLS 1.3 sessions with 0RTT without changing their application code.

A high performance server for HTTP, reverse proxy, mail, ...

We replace OpenSSL with miTLS & its crypto: the modified server supports TLS 1.3 with tickets and 0-RTT requests.
Concrete Applications

QUIC is an IETF Working Group that is chartered to deliver the next transport protocol for the Internet.

See our contribution guidelines if you want to work with us.

Upcoming Meetings
Our next meeting is at IETF 101 in London, with an interop during the Hackathon.

Our Documents
Our initial documents cover three different aspects of the ‘core’ QUIC protocol, and also define a mapping of HTTP semantics to it.
Modelling concrete cryptographic security in F*
Cryptographic Integrity: Message Authentication Codes (MAC)

This plain interface says nothing about the security of MACs!
Cryptographic Integrity: UF-CMA security (1/3)

```
module HMAC_SHA256

type key

type msg = bytes

type tag = lbytes 32

val log: mem → key → seq (msg × tag) (* ghost *)

val keygen: unit → ST key
  (ensures λ h₀ k h₁ → log h₁ k = empty)

val mac: k:key → m:msg → ST tag
  (ensures λ h₀ t h₁ → log h₁ k = log h₀ k ++ m ⊸ t)

val verify: k:key → m:msg → t:tag → bool
  (ensures λ h₀ b h₁ → b = mem (log h₀ k) (m ⊸ t))
```
Idealization

In cryptography, most security properties are defined in terms of indistinguishability games between an ideal and a real functionality

- Flip a coin $b$
- Let $F$ be the ideal function if head, the real one if tail
- Let $A$ be a program that can call $F$ and tries to guess $b$
- The advantage of $A$ is $|Pr(A() = b) - 1/2|$
- The functionality of $\varepsilon$-secure if $Adv(A) \leq \varepsilon$
Our ideal interface reflects the security of a chosen-message game [Goldwasser ’88].

The MAC scheme is \( \epsilon \)-UF-CMA-secure against a class of probabilistic, computationally bounded attackers when the game returns true with probability at most \( \epsilon \).

**UF-CMA programmed in F***

```
let game attacker =
  let k = MAC.keygen() in
  let log = ref empty in

let oracle msg =
  log := !log ++ msg;
  MAC.mac k msg in

let msg, forgery = attacker oracle in
MAC.verify k msg forgery &&
not (Seq.mem msg !log)
```

**Game** \( \text{Mac1}^b(MAC) \)

\[
\begin{align*}
  k & \leftarrow^{\$} \text{MAC.keygen}(\varepsilon); \\
  \log & \leftarrow \bot \\
  \text{return } \{\text{Mac, Verify}\}
\end{align*}
\]

**Oracle** \( \text{Mac}(m) \)

\[
\begin{align*}
  t & \leftarrow \bot \\
  \text{if } b \land r \\
  \text{then } t \leftarrow^{\$} \text{byte}^\text{MAC.} \langle t \rangle \\
  \text{return } t
\end{align*}
\]
Cryptographic Integrity: UF-CMA security (3/3)

Ideal system:
- Mac
  - Real interface
- Ideal filter
  - Ideal MAC
  - RPC protocol
    - Secure RPC
- Any p.p.t. adversary
  - Perfectly safe by typing

Concrete system:
- Mac
  - Real interface
- RPC protocol
  - Sample protocol typed against ideal MAC interface
- Any p.p.t. adversary
  - Safe too, with probability $1 - \epsilon$

Concrete algorithm assumed UF-CMA computationally safe too,

$\epsilon$
Cryptographic Integrity: Two styles for ideal MACs

**module MAC (* stateful *)**

**type** key

**val** log: mem → key → Seq msg

**val** keygen:
unit → ST key
(ensures \( \lambda h_0 \ k \ h_1 \rightarrow \)
\( \log h_1 \ k = \text{empty} \))

**val** mac:
k: key → m: msg → ST tag
(ensures \( \lambda h_0 \ t \ h_1 \rightarrow \)
\( \log h_1 \ k = \log h_0 \ k ++ m \))

**val** verify:
k: key → m: msg → t: tag → ST bool
(ensures \( \lambda h_0 \ b \ h_1 \rightarrow \)
\( b \implies \text{mem} (\log h_0 \ k) \ m \))

**module MAC (* logical *)**

**type** property: msg → Type

**type** key (p: prop)

**val** keygen:
\( \#p: \text{property} \rightarrow \text{St} \ (\text{key} \ p) \)

**val** mac:
\( \#p: \text{property} \rightarrow \text{key} \ p \rightarrow \)
m: msg \{p \ m\} \rightarrow \text{St} \ tag

**val** verify:
\( \#p: \text{property} \rightarrow \text{key} \ p \rightarrow m : \text{msg} \rightarrow \text{tag} \rightarrow \)
\text{St} (b: bool \{b \implies p \ m\})

(* proof idea: maintain a private stateful log: *)

**type** log (p: property) =
mref (seq (m: msg \{p\})) grows
Authenticated Encryption
We rely on type abstraction:
Ideal encryption never accesses the plaintext, is info-theoretically secure.
Authenticated Encryption: Game-based security assumption

We program this game in F* parameterized by a real scheme AE and the flag b.

\[
\text{Game } \text{Ae}(\mathcal{A}, \text{AE})
\]
\[
b \leftarrow \{0, 1\}; \ L \leftarrow \emptyset; \ k \leftarrow \text{AE.keygen}()
\]
\[
b' \leftarrow A^{\text{Encrypt,Decrypt}}(); \ \text{return } (b \equiv b')
\]

Oracle Encrypt(p)
if b then c \leftarrow \text{byte}^{\ell_c}; \ L[c] \leftarrow p
else c \leftarrow \text{AE.encrypt} k p
return c

Oracle Decrypt(c)
if b then p \leftarrow L[c]
else p \leftarrow \text{AE.decrypt} k c
return p

**Definition 1 (AE-security):** Given AE, let \( \epsilon_{\text{AE}}(\mathcal{A}[q_e, q_d]) \) be the advantage of an adversary \( \mathcal{A} \) that makes \( q_e \) queries to Encrypt and \( q_d \) queries to Decrypt in the \( \text{Ae}^b(AE) \) game.

We capture its security using types to keep track of the content of the log.
Scaling Up: Authenticated Encryption for TLS

Same modelling & verification approach
concrete security: each lossy step documented by a game and a reduction (or an assumption) on paper

Standardized complications
- multiple algorithms and constructions (crypto agility)
- multiple keys
- conditional security (crypto strength, compromise)
- wire format, fragmentation, padding
- stateful (stream encryption)

Poor TLS track record
- Many implementation flaws
- Attacks on weak cryptography (MD5, SHA1, ...)
- Attacks on weak constructions (MAC-Encode-then-Encrypt)
- Attacks on compression
- Persistent side channels
- Persistent truncation attacks
The TLS Record Layer

<table>
<thead>
<tr>
<th>Handshake</th>
<th>AppData</th>
<th>Alert</th>
<th>Plaintext</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Key 0 (1 sided)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Key 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Key 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Key 3</td>
</tr>
</tbody>
</table>

**Write channel**

**Read channel**
TLS 1.3 gets rid of weak constructions, encrypts parts of the handshake, introduces plenty of auxiliary keys.
The TLS Record Layer (F*)

We model record-layer security using a game at every level of the construction.

We make code-based security assumptions on the crypto primitives (PRF, MAC)

We obtain security guarantees at the top-level API for the TLS record layer
Stream Encryption: Security Definition

plaintext message

plaintext fragments

Attack at dawn!

encrypt

#2

Attack at
dawn!


TLS record layer

untrusted network

3ef87abce4363

a3b1684fbc770

...
Stream Encryption: Security Definition

Attack at dawn!

Client

established connection (keys)

Server

plaintext message

plaintext fragments

Attack at dawn!

encrypt

random sampling

ideal encryption log

#0 3ef87abce4363
#1 a3b1684fbc770

Attack at dawn!

untrusted network

ciphertext fragments

3ef87abce4363
a3b1684fbc770

decrypt

table lookup

3ef87abce4363
One-time Pad for the MAC

CHACHA20_POLY130 5 for the TLS record
Stream Encryption: Assumptions

One-Time MACs (INT-CMA1)

\[
\text{Game } \text{UF-1CMA}(A, \text{MAC}) \quad \begin{align*}
    k &\leftarrow \text{MAC.keygen}(\varepsilon); \log \leftarrow \bot \\
    (m^*, t^*) &\leftarrow A_{\text{Mac}}^\text{Mac} \\
    \text{return } \text{MAC.verify}(k, m^*, t^*) \\
    \land \log \neq (m^*, t^*)
\end{align*}
\]

\[
\text{Oracle } \text{Mac}(m) \quad \begin{align*}
    \text{if } \log \neq \bot &\text{ return } \bot \\
    t &\leftarrow \text{MAC.mac}(k, m) \\
    \log &\leftarrow (m, t) \\
    \text{return } t
\end{align*}
\]

Ciphers (IND-PRF)

\[
\text{Game } \text{Prf}^b(\text{PRF}) \quad \begin{align*}
    T &\leftarrow \emptyset \\
    k &\leftarrow \text{PRF.keygen()} \\
    T[0] &\leftarrow \text{PRF.eval}(k, m) \\
    \text{return } \{\text{Eval}\}
\end{align*}
\]

\[
\text{Oracle } \text{Eval}(m) \quad \begin{align*}
    \text{if } T[m] = \bot &\text{ return } \bot \\
    \text{if } b &\text{ then } T[m] \leftarrow \text{byte}^b \\
    \text{else } T[m] &\leftarrow \text{PRF.eval}(k, m) \\
    \text{return } T[m]
\end{align*}
\]

For both GF128 or Poly1305, we get strong probabilistic security.

Assumed for AES and Chacha20
Stream Encryption: Assumptions

One-Time MACs (INT-CMA1)

\[
\text{Game UF-1CMA}(A, \text{MAC}) \quad \begin{align*}
    k & \xleftarrow{\$} \text{MAC.keygen}(\varepsilon); \\
    \log & \xleftarrow{\$} \perp \\
    (m^*, t^*) & \xleftarrow{\$} A^\text{Mac} \\
    \text{return } & \text{MAC.verify}(k, m^*, t^*) \\
    & \land \log \neq (m^*, t^*)
\end{align*}
\]

Oracle Mac(m)

\[
\begin{align*}
    \text{if } \log \neq \perp & \text{ return } \perp \\
    t & \xleftarrow{\$} \text{MAC.mac}(k, m) \\
    \log & \xleftarrow{\$} (m, t) \\
    \text{return } t
\end{align*}
\]

Construction:
authenticated materials and their lengths are encoded as coefficients of a polynomial in a field (GF128 or 2^130 -5)
The MAC is the polynomial evaluated at a random point, then masked.
We get strong probabilistic security.

Ciphers (IND-PRF)

\[
\text{Game Prf^b(PRF)} \quad \begin{align*}
    T & \xleftarrow{\$} \emptyset \\
    k & \xleftarrow{\$} \text{PRF.keygen()} \\
    t & \xleftarrow{\$} \text{PRF.mac}(k, m) \\
    \text{return } & \{\text{Eval}\}
\end{align*}
\]

Oracle Eval(m)

\[
\begin{align*}
    \text{if } T[m] = \perp & \text{ return } \perp \\
    \text{if } b & \text{ then } T[m] \xleftarrow{\$} \text{byte}^b \\
    \text{else } T[m] & \xleftarrow{\$} \text{PRF.eval}(k, m) \\
    \text{return } T[m]
\end{align*}
\]

Modelling:
we use a variant with specialized oracles for each usage of the resulting blocks
- as one-time MAC key materials
- as one-time pad for encryption
- as one-time pad for decryption
Stream Encryption: Construction

Given
- a cipher, modelled as a pseudo-random function
- a field for computing one-time MACs
- injective message encodings

We program and verify a generic authenticated stream encryption with associated data.

We show
- safety
- functional correctness
- security (reduction to PRF assumption)
- concrete security bounds for the 3 main record ciphersuites of TLS

TLS-specific mechanisms
- fragmentation
- content multiplexing
- length-hiding, padding
- re-keying
- 0-RTT, 0.5-RTT

many kinds of proofs not just code safety!
Stream Encryption: Concrete Bounds

Theorem: the 3 main record ciphersuites for TLS 1.2 and 1.3 are secure, except with probabilities

<table>
<thead>
<tr>
<th>Ciphersuite</th>
<th>$\epsilon_{\text{Lhse}}(A[q_e, q_d]) \leq$</th>
</tr>
</thead>
<tbody>
<tr>
<td>General bound</td>
<td>$\epsilon_{\text{Prf}}(B[q_e(1 + \lceil (2^{14} + 1)/\ell_b \rceil) + q_d + j_0]) + \epsilon_{\text{MMac1}}(C[2^{14} + 1 + 46, q_d, q_e + q_d])$</td>
</tr>
<tr>
<td>ChaCha20-Poly1305</td>
<td>$\epsilon_{\text{Prf}}(B[q_e\left(1 + \left\lceil \frac{(2^{14}+1)}{64} \right\rceil \right) + q_d]) + \frac{q_d}{2^{93}}$</td>
</tr>
<tr>
<td>AES128-GCM</td>
<td>$\epsilon_{\text{Prp}}(B[q_b]) + \frac{q_b^2}{2^{129}} + \frac{q_d}{2^{118}}$</td>
</tr>
<tr>
<td>AES256-GCM</td>
<td>$\epsilon_{\text{Prp}}(B[2^{34.5}]) + \frac{1}{2^{60}} + \frac{1}{2^{56}}$</td>
</tr>
<tr>
<td>AES128-GCM</td>
<td>$q_b = q_e(1 + \lceil (2^{14} + 1)/16 \rceil) + q_d + 1$</td>
</tr>
<tr>
<td>AES128-GCM</td>
<td>$q_b = q_e(1 + \lceil (2^{14} + 1)/16 \rceil) + q_d + 1$</td>
</tr>
</tbody>
</table>

$q_e$ is the number of encrypted records;
$q_d$ is the number of chosen-ciphertext decryptions;
$q_b$ is the total number of blocks for the PRF

Probabilistic proof (on paper) in abstract field + $F^*$ verification

$\epsilon_{\text{MMac1}} = \frac{d \cdot \tau \cdot q_v}{|R|}$

Standard crypto assumption

$\epsilon_{\text{Prf}}$

IND-PRF

IND-1CMA

AEAD

AEAD.Encoding

AEAD.Invariant

Stream Encryption

$\epsilon_{\text{Lhse}}(A[q_e, q_d])$

$= \epsilon_{\text{Prf}} + \epsilon_{\text{MMac1}}$

$F^*$ type-based verification on code formalizing game-based reduction
# Stream Encryption: Verification Effort

<table>
<thead>
<tr>
<th>Module Name</th>
<th>Verification Goals</th>
<th>LoC</th>
<th>% annot</th>
<th>ML LoC</th>
<th>C LoC</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>StreamAE</td>
<td>Game StAE\textsuperscript{b} from §VI</td>
<td>318</td>
<td>40%</td>
<td>354</td>
<td>N/A</td>
<td>307s</td>
</tr>
<tr>
<td>AEADProvider</td>
<td>Safety and AEAD security (high-level interface)</td>
<td>412</td>
<td>30%</td>
<td>497</td>
<td>N/A</td>
<td>349s</td>
</tr>
<tr>
<td>Crypto.AEAD</td>
<td>Proof of Theorem 2 from §V</td>
<td>5,253</td>
<td>90%</td>
<td>2,738</td>
<td>2,373</td>
<td>1,474s</td>
</tr>
<tr>
<td>Crypto.Plain</td>
<td>Plaintext module for AEAD</td>
<td>133</td>
<td>40%</td>
<td>95</td>
<td>85</td>
<td>8s</td>
</tr>
<tr>
<td>Crypto.AEAD.Encoding</td>
<td>AEAD encode function from §V and injectivity proof</td>
<td>478</td>
<td>60%</td>
<td>280</td>
<td>149</td>
<td>708s</td>
</tr>
<tr>
<td>Crypto.Symmetric.PRF</td>
<td>Game PrfC\textsuperscript{b} from §IV</td>
<td>587</td>
<td>40%</td>
<td>522</td>
<td>767</td>
<td>74s</td>
</tr>
<tr>
<td>Crypto.Symmetric.Cipher</td>
<td>Agile PRF functionality</td>
<td>193</td>
<td>30%</td>
<td>237</td>
<td>270</td>
<td>65s</td>
</tr>
<tr>
<td>Crypto.Symmetric.AES</td>
<td>Safety and correctness w.r.t pure specification</td>
<td>1,254</td>
<td>30%</td>
<td>4,672</td>
<td>3,379</td>
<td>134s</td>
</tr>
<tr>
<td>Crypto.Symmetric.Chacha20</td>
<td>Safety and correctness w.r.t pure specification</td>
<td>965</td>
<td>80%</td>
<td>296</td>
<td>119</td>
<td>826s</td>
</tr>
<tr>
<td>Crypto.Symmetric.UFICMA</td>
<td>Game MMac\textsuperscript{b} from §III</td>
<td>617</td>
<td>60%</td>
<td>277</td>
<td>467</td>
<td>428s</td>
</tr>
<tr>
<td>Crypto.Symmetric.MAC</td>
<td>Agile MAC functionality</td>
<td>488</td>
<td>50%</td>
<td>239</td>
<td>399</td>
<td>387s</td>
</tr>
<tr>
<td>Crypto.Symmetric.GF128</td>
<td>(GF(128)) polynomial evaluation and GHASH encoding</td>
<td>306</td>
<td>40%</td>
<td>335</td>
<td>138</td>
<td>85s</td>
</tr>
<tr>
<td>Crypto.Symmetric.Poly1305</td>
<td>(GF(2^{130} - 5)) polynomial evaluation and Poly1305 encoding</td>
<td>604</td>
<td>70%</td>
<td>231</td>
<td>110</td>
<td>245s</td>
</tr>
<tr>
<td>Hacl.Bignum</td>
<td>Bignum library and supporting lemmas for the functional correctness of field operations</td>
<td>3,136</td>
<td>90%</td>
<td>1,310</td>
<td>529</td>
<td>425s</td>
</tr>
<tr>
<td>FStar.Buffer.\textsuperscript{*}</td>
<td>A verified model of mutable buffers (implemented natively)</td>
<td>1,340</td>
<td>100%</td>
<td>N/A</td>
<td>N/A</td>
<td>563s</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td>15,480</td>
<td>78%</td>
<td>12,083</td>
<td>8,795</td>
<td>1h 41m</td>
</tr>
</tbody>
</table>
Verified High-Assurance Crypto Libraries

Network buffers

Crypto Algorithms

**RSA**

**ECDH**

**SHA**

**4Q**

**X.509**

**ASN.1**

**HTTPS**

**TLS**
Design goals

- Low-level implementations
- Functional correctness wrt pure specification
- Runtime safety (e.g. memory safety)
- Side-channel resistance
Does functional correctness matter?

• Bugs happen: 3 fresh ones just in OpenSSL’s poly1305.

"These produce wrong results. The first example does so only on 32 bit, the other three also on 64 bit."

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

“I'm probably going to write something to generate random inputs and stress all your other poly1305 code paths against a reference implementation.”
Does memory safety matter?

- Most real-world vulnerabilities are memory safety errors.
- Program verification tools often use memory-managed functional languages.
- These languages are too slow, and GC introduces new side channels that are hard to mitigate.
1. Compile restricted subset of verified source code to efficient C/C++ ; or
2. Use a DSL for portable verified assembly code
Sample crypto algorithm in OpenSSL

- Hand-crafted mix of Perl and assembly
- Customized for 50+ hardware platforms
- Why?
  Performance!
  several bytes/cycle

---

```
sub BODY_00_15 {
my ($i,$a,$b,$c,$d,$e,$f,$g,$h) = @_;
$code.='<<___ if ($i<16);
  #if __ARM_ARCH__>=7
  @ ldr  $t1,[inp],#4 @ $i
  # if $i==15
  str  $inp,[sp,#17*4] @ make room for $t4
  # endif
  eor  $t0,$e,$e,ror`$Sigma1[1]-$Sigma1[0]`
  add  $a,$a,$t2 @ h+=Maj(a,b,c) from the past
  eor  $t0,$t0,$e,ror`$Sigma1[2]-$Sigma1[0]` @ Sigma1(e)
  # ifndef __ARMELB__
  rev  $t1,$t1
  # endif
  #else
  @ ldrb $t1,[inp,#3] @ $i
  add  $a,$a,$t2 @ h+=Maj(a,b,c) from the past
  ldrb $t2,[inp,#2]
  ldrb $t0,[inp,#1]
  orr  $t1,$t1,$t2,lsl#8
  ldrb $t2,[inp],#4
  orr  $t1,$t1,$t0,lsl#16
  # if $i==15
  str  $inp,[sp,#17*4] @ make room for $t4
  # endif
  eor  $t0,$e,$e,ror`$Sigma1[1]-$Sigma1[0]`
  orr  $t1,$t1,$t2,lsl#24
  eor  $t0,$t0,$e,ror`$Sigma1[2]-$Sigma1[0]` @ Sigma1(e)
  # endif
```
```
Sample crypto algorithm: poly1305

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

Authenticate data by
1. Encoding it as a polynomial in the prime field $2^{130} - 5$
2. Evaluating it at a random point: the first part of the key $k$
3. Masking the result using the second part of the key $m$
Sample crypto algorithm: poly1305

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

**Security?**
If the sender and the receiver disagree on the data \( \vec{w} \) then the difference of their polynomials is not null.

Its evaluation at a random \( k \) is 0 with probability \( \approx \frac{|\vec{w}|}{2^{130}} \).
Sample F* code: the spec for the multiplicative MAC used in TLS 1.3

Its verified optimized implementation for x64 takes 3K+ LOCs
Sample crypto algorithm: poly1305

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

A typical 64-bit arithmetic implementation:
1. Represent elements of the prime field for \( p = 2^{130} - 5 \) using 3 limbs holding 42 + 44 + 44 bits in 64-bit registers
2. Use \((a \cdot 2^{130} + b) \mod p = (a + 4a + b) \mod p\) for reductions
3. Unfold loop
Low*: low-level programming in F*

We must get to Low* after typing, erasure, and much inlining

- Compile-time error otherwise
- Goal: zero implicit heap allocations
- Non-goal: bootstrapping and high-level modelling (we have F*/OCaml for that)

Machine arithmetic

- Static checks for overflows
- Explicit coercions

Not the usual ML memory

Infix pointer arithmetic (erased lengths)

Static tracking of
- Liveness & index ranges
- Stack allocation
- Manual allocation
- Regions

No F* hack! Just libraries.
Low*: a subset of F* for safe C-style programming

Supports compilation to C, in nearly 1-1 correspondence, for auditability of our generated code.

Features a C-like view of memory (pointer arithmetic with verified safety)

KreMLin: a new compiler from Low* to C (ICFP’17)

- Semantics preserving from Low* to CompCert Clight
- Also: does not introduce memory-based side channels
- Then compile C using mainstream compilers
- Or, CompCert
KreMLin: from F* to Low* to C* to C

• Why C/C++ ???
  Performance, portability
  Predictability (GC vs side channels)
  Interop (mix’n match)
  Readability, transparency
    (code review)
  Adoption, maintenance

• Formal translations

• Various backends
  Clang/LLVM; gcc
  CompCert, with verified translation
    from C* to Clight

• What KreMLin does
  Monomorphization of dependent types
  Data types to flat tagged unions
  Compilation of pattern matching
  From expressions to statements (hoisting)
  Name-disambiguation (C’s block-scoping)
  Inlining (in-scope closures, stackInline)

• Early results for HACL*:
  high assurance crypto library
  15 KLOCs of type-safe, partially-verified elliptic curves, symmetric encryption...
  Up to 150x speedup/ocamlopt
  Down by 50% vs C/C++ libraries
Low* Poly1305 compiled to C

```haskell
val poly1305_last_pass : 
  acc:felem →
  Stack unit
  (requires (λ h → live h acc ∧ bounds (as_seq h acc) p44 P44 P44))
  (ensures (λ h0 h1 → live h0 acc ∧ bounds (as_seq h0 acc) p44 P44 P44)
               ∧ live h1 acc ∧ bounds (as_seq h1 acc) p44 P44 P44)
  ∧ modifies_1 acc h0 h1
  ∧ as_seq h1 acc == Hacl.Spec.Poly1305_64.poly1305_last_pass_spec_ (as_seq h0 acc))

let poly1305_last_pass_acc =
  let a0 = acc.(0ul) in
  let a1 = acc.(1ul) in
  let a2 = acc.(2ul) 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_1 (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'
```

```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 t10 = acc[0];
    uint64_t t11 = acc[1];
    uint64_t t0 = i0 & (((uint64_t)1) << (uint32_t)44) - (uint64_t)1);
    uint64_t t1 = 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 t0 = FStar_Uint64_gte_mask(a00, (uint64_t)0xfffffffff);
    uint64_t t1 = FStar_Uint64_eq_mask(a1, (uint64_t)0xfffffffff);
    uint64_t t2 = FStar_Uint64_eq_mask(a2, (uint64_t)0x3ffffff);
    uint64_t t0 = mask0 & mask1 & mask2;
    uint64_t a0 = a00 - (uint64_t)0xfffffffff & mask;
    uint64_t a1 = a1 - (uint64_t)0xfffffffff & mask;
    uint64_t a2 = a2 - (uint64_t)0xfffffffff & mask;
    acc[0] = a0;
    acc[1] = a1;
    acc[2] = a2;
}
```
Performance for verified C code compiled from F* as fast as best hand-written portable C implementations.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>HACL*</th>
<th>OpenSSL</th>
</tr>
</thead>
<tbody>
<tr>
<td>ChaCha20</td>
<td>6.17 cy/B</td>
<td>8.04 cy/B</td>
</tr>
<tr>
<td>Poly1305</td>
<td>2.07 cy/B</td>
<td>2.16 cy/B</td>
</tr>
<tr>
<td>Curve25519</td>
<td>157k cy/mul</td>
<td>359k cy/mul</td>
</tr>
</tbody>
</table>

Still slower than best hand-written assembly language implementations.
Vale: extensible, automated assembly language verification (Usenix’17)

Instructions

```
type reg = r0 | r1 | ...
type ins =
    Mov(dst:reg, src:reg)
    | Add(dst:reg, src:reg)
    | Neg(dst:reg)
    ...
```

Semantics

```
eval(Mov(dst, src), ...) = ...
eval(Add(dst, src), ...) = ...
eval(Neg(dst), ...) = ...
...
```

code generation

```
print(Mov(dst, src), ...) = "mov " + (...dst) + (...src)
print(Add(dst, src), ...) = ...
```

Vale source code

```
machine interface
procedure mov(...) requires ...
ensures ...
{ ... }
procedure add(...) ...
```

Program

```
procedure quadruple(...) requires 0 <= r0 < 2^30;
ensures r1 == r0 * 4;
{     mov(r1, r0);
     add(r1, r0);
     add(r1, r1);
    }
```

Trusted Computing Base

```
Trust the correctness & side-channel protection
```
procedure poly1305_reduce()
{
    
    And64(rax, d3);
    Mov64(h2, d3);
    Shr64(d3, 2);
    And64(h2, 3);
    Add64Wrap(rax, d3);
    Add64Wrap(h0, rax);
    Adc64Wrap(h1, 0);
    Adc64Wrap(h2, 0);
    ...
}
procedure poly1305_reduce() returns(ghost hOut:int)

let
    n := 0x1_0000_0000_0000_0000;
    p := 4 * n * n - 5;
    hIn := (n * n) * d3 + n * h1 + h0;
    d3 @= r10; h0 @= r14; h1 @=rbx; h2 @= rbp;
modifies
    rax; r10; r14; rbx; rbp; efl;
requires
    d3 / 4 * 5 < n;
    rax == n - 4;
ensures
    hOut % p == hIn % p;
    hOut == (n * n) * h2 + n * h1 + h0;
    h2 < 5;
{
    lemma_BitwiseAdd64();
    lemma_poly_bits64();
    And64(rax, d3); Mov64(h2, d3);
    Shr64(d3, 2);
    And64(h2, 3);
    Add64Wrap(rax, d3);
    Add64Wrap(h0, rax);
    Add64Wrap(h1, 0);
    Adc64Wrap(h2, 0);
    Adc64Wrap(h2, 0);
}
Performance: **OpenSSL vs. Vale**

- AES: OpenSSL with SIMD, AES-NI
- Poly1305 and SHA-256: OpenSSL non-SIMD assembly language (same assembly for OpenSSL, Vale)
Verified interoperability between Low* and Vale
(Sneak peek of work in progress)

Goals: **End-to-end** functional correctness and side-channel resistance

- Reconcile the memory models of Low* and Vale

(Each of these regions maps references to values)

- Compose the secret-independent trace theorems of Low* and Vale

- Low* has a structured memory model

- Vale memory is a flat array of bytes

Enhance the Low* memory model to also have a flat array of bytes view and update it, **transparently**

Reflect changes performed by the Vale code in the structured view, *allowing for temporary inconsistencies*
TLS 1.3 Handshake (Outline)
TLS protocol overview

<table>
<thead>
<tr>
<th>Client</th>
<th>Server</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hello</td>
<td>Protocol negotiation</td>
</tr>
<tr>
<td></td>
<td>- Agree on version</td>
</tr>
<tr>
<td></td>
<td>- Agree on ciphersuite</td>
</tr>
<tr>
<td></td>
<td>- Determines all crypto algos</td>
</tr>
<tr>
<td>Keying</td>
<td>Authenticated Key Exchange</td>
</tr>
<tr>
<td></td>
<td>- Verify server/client identity</td>
</tr>
<tr>
<td></td>
<td>- Generate master secret</td>
</tr>
<tr>
<td></td>
<td>- Derive connection keys</td>
</tr>
<tr>
<td>Finished</td>
<td>Key &amp; transcript confirmation</td>
</tr>
<tr>
<td></td>
<td>- Completes authentication</td>
</tr>
<tr>
<td></td>
<td>- Matches transcripts</td>
</tr>
<tr>
<td></td>
<td>- Authenticated encryption</td>
</tr>
<tr>
<td>AppData</td>
<td>Application data streams</td>
</tr>
<tr>
<td></td>
<td>- Full duplex channel</td>
</tr>
<tr>
<td></td>
<td>- Authenticated encryption</td>
</tr>
</tbody>
</table>
Programming & Verifying the TLS Handshake

Handshake

- Negotiation
- Configuration Certificates
- Signing

Syntax: parse/format

Messages
Extensions

Session Log

flights
digests
Hash

Key Schedule

keys
shares
OD
H
PRF

State machine

config & mode

HMAC

control

TLS API

Application (HTTPS etc)

Network (TCP)

Record Layer Protection

issue fresh keys

send & receive messages
Low-level parsing and formatting

Most of the RFC, most of the code.

Correctness?

**Metaprogramming in F***

Performance?

Intermediate copies considered harmful.

Security?

Handshake digest computed on the fly

Example: `ClientHello` message

Example: `HandshakeLog.recv`
**high-level parser**

val parseCH: bytes ->
   option clientHello

**inverse properties**

val injCH: clientHello ->
   Lemma ...

**low-level validator**

val validateCH: len: Uint32.t ->
   input: lbuffer len ->
   Stack (option (erased clientHello * Uint32.t))
   (requires fun h0 -> live input)
   (ensures fun h0 result h1 ->
      h0 = h1 /
      match result with
      | Some (ch, pos) ->
        pos <= len /
        format ch = buffer.read input h0 0..pos-1
      | None -> True)

**high-level type**

type clientHello =
   | ClientHello:
     pv: protocolVersion ->
     id: vlbytes1 0 32 ->
     cs: seq ciphersuite {...} -> ...

**high-level formatter**

val formatCH:
   clientHello -> bytes

**low-level in-place code extracted to C**

```
struct {
  ProtocolVersion legacy_version = 0x0303; /* TLS v1.2 */
  Random random;
  opaque legacy_session_id<0..32>;
  CipherSuite cipher_suites<2..2^16-2>;
  opaque legacy_compression_methods<1..2^8-1>;
  Extension extensions<8..2^16-1>;
} ClientHello;
```
Low-level parsing: variable-length bytes

e.g. session_id <0..32> is formatted as a “vlbytes 1”

```haskell
let parse_vlbytes₁ (#t::Type₀) (p::parser t)::parser t =
parse_u₈ `and_then` (λ len → parse_sized₁ p len)
```
Circular problem: secure negotiation relies on the crypto algorithms and keys being negotiated

An attacker may cause honest participants to agree on weak or mismatched parameters.

TLS 1.3 adopted our recommendations to defend against downgrade attacks (modelled at Oakland’16):

Simple verification (ghost handshake digests)
**Handshake State Machine**

### TLS 1.2 (Full Handshake)

- **ClientHello**
- **ServerHello**
  - Certificate
  - ServerKeyExchange
  - CertificateRequest
  - ServerHelloDone

- **Certificate***
- **Finished**
- **Application Data***

### TLS 1.2 (Abbreviated Handshake)

- **ClientHello**
- **ServerHello**
  - [ChangeCipherSpec]
  - Finished

- **Application Data***

### TLS 1.3 (Full Handshake)

- **ClientHello** + key_share
  - ServerHello + key_share
  - {EncryptedExtensions}
  - {CertificateRequest}*
  - {Certificate}*
  - {CertificateVerify}*
  - {Finished}
  - {Certificate}*
  - {CertificateVerify}*
  - {Finished}
  - [Application Data]`

### TLS 1.3 (PSK Handshake with 0RTT)

- **ClientHello** + key_share*
  - psk_key_exchange_modes
  - pre_shared_key
  - (Application Data*)

- **ServerHello** + pre_shared_key
  - + key_share*
  - {EncryptedExtensions}
  - {Finished}

- {Certificate}*
  - {CertificateVerify}*
  - [Application Data]`

### TLS 1.3 (incorrect key share)

- **ClientHello** + key_share
  - HelloRetryRequest

- **ClientHello** + key_share
  - ...

### TLS 1.2 (Abbreviated Handshake)

- **ClientHello**
  - **ServerHello**
    - [ChangeCipherSpec]
    - Finished

- **Application Data***
Handshake State Machine

Boxes represent successive flights. Gray states need not be represented.
Caption: two kinds of key derivation steps

key materials

prior secret

Extraction

prior secret

transcript

Extraction

new secret

derived secret

Diffie-Hellman shared secret \((g^{xy})\)

early secret

handshake secret

master secret

pre-shared keys for future sessions

Encryption

Integrity

Export (QUIC)

Handshake Integrity

Encryption

Integrity

Encryption

Export (QUIC)

pre-shared key

Encryption

Integrity

Export (QUIC)

Pre-secret

derived secret
Diffie-Hellman shared secret ($g^{xy}$)

Our (fresh) crypto model precisely reflects F* code modularity, involves a security definition for each color, supports agility and key compromise.
Exercise:
RSA in $F^*$

$$N = p \times q \quad \text{(two primes)}$$
$$e \times d = 1 \quad [\varphi(N)]$$

$$Sign(m, (N, d)) = m^d \quad [N]$$

1. Simple specification
2. Fast exponentiation: square & multiply
3. Blinded implementation
   (for side-channel resistance)
Exercise: RSA in $\mathbb{F}^*$

$N = p \times q \quad \text{(two primes)}$

$e \times d = 1 \quad [\varphi(N)]$

$\text{Sign}(m, (N, d)) = m^d \quad [N]$

$m' = m \, r^e \quad \text{(where } r \text{ and } N \text{ are coprime)}$

$s' = m^d \, r^{ed} \quad [N]$

$\text{Sign}(m, (N, d)) = s' \, r^{-1} \quad [N]$
Everest: verified drop-in replacements for the HTTPS ecosystem

• complex, critical, verifiable
• close collaboration: crypto, system, compilers, verification
• new tools: F*, KreMLin, Vale
• safety, functional correctness & crypto security for standard-compliant system code

Code, papers, details at
https://project-everest.github.io
https://github.com/project-everest
https://mitls.org
https://www.fstar-lang.org