Authenticated Encryption in TLS
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

Handshake

AppData

Alert

Plaintext

Key 0 (1 sided)

Key 1

Key 2

Key 3

Write channel

Handshake

AppData

Alert

Plaintext

Key 1

Key 2

Key 3

Read channel
TLS 1.3 gets rid of weak constructions, encrypts parts of the handshake, introduces plenty of auxiliary keys.

The TLS Record Layer (TLS 1.3)
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
Crypto security for TLS Stream Encryption

1. Security definition
2. TLS constructions (AEAD)
3. Concrete security bounds
4. Verification
5. Performance
Stream Encryption: Security Definition

plaintext message

plaintext fragments

Attack at dawn!

encrypt

#2

TLS record layer

ciphertext fragments

3ef87abce4363

a3b1684fbc770

untrusted network

Client

established connection (keys)

Server

Attack at

decrypt

#1

3ef87abce4363
Stream Encryption: Security Definition

plaintext message

Attack at dawn!

... established connection (keys) ...

Client

Server

evaluated encryption log

#0 3ef87abce4363 3ef87abce4363

#1 a3b1684fbc770 Attack at dawn!

... table lookup ...

ciphertext fragments

3ef87abce4363

a3b1684fbc770

... untrusted network ...
Game \text{RoR}(AEAD)
\begin{align*}
\text{log} & \leftarrow \emptyset; \\
\text{b} & \leftarrow \{0, 1\} \\
\text{k} & \leftarrow \text{keygen()} \\
\text{b}' & \leftarrow \mathcal{A}_{\text{Encrypt,Decrypt}}() \\
\text{return } (b' = b)
\end{align*}

\text{Oracle Encrypt}(n, a, m)
\begin{align*}
\text{if } \text{log}[n] \neq \bot & \text{ return } \bot \\
\text{if } b & \\
& \quad c \leftarrow \text{Byte}^{m+\text{taglen}} \\
\text{else} & \\
& \quad c \leftarrow \text{encrypt}(k, n, a, m) \\
\text{log}[n] & \leftarrow (a, m, c) \\
\text{return } c
\end{align*}

\text{Oracle Decrypt}(n, a, c)
\begin{align*}
\text{if } b = 1 & \\
& \quad \text{if } \text{log}[n] = (a, m, c) \text{ return } \text{Some}(m) \\
& \quad \text{else return None} \\
\text{else return } \text{decrypt}(k, n, a, c)
\end{align*}
We program & verify AEAD for TLS 1.2 and TLS 1.3.

We do not consider here classic, time-battered TLS modes such as AES_CBC (Mac-Encode-then-Encrypt).

**Stream encryption in TLS 1.3**

### A.4. Cipher Suites

A symmetric cipher suite defines the pair of the AEAD algorithm and hash algorithm to be used with HKDF. Cipher suite names follow the naming convention:

```
CipherSuite TLS_AEAD_HASH = VALUE;
```

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS</td>
<td>The string “TLS”</td>
<td>(0x13,0x01)</td>
</tr>
<tr>
<td>AEAD</td>
<td>The AEAD algorithm used for record protection</td>
<td>(0x13,0x02)</td>
</tr>
<tr>
<td>HASH</td>
<td>The hash algorithm used with HKDF</td>
<td>(0x13,0x03)</td>
</tr>
<tr>
<td>VALUE</td>
<td>The two byte ID assigned for this cipher suite</td>
<td>(0x13,0x04)</td>
</tr>
</tbody>
</table>

This specification defines the following cipher suites for use with TLS 1.3.

**Example Cipher Suites**

- **TLS_AES_128_GCM_SHA256**: (0x13,0x01)
- **TLS_AES_256_GCM_SHA384**: (0x13,0x02)
- **TLS_CHACHA20_POLY1305_SHA256**: (0x13,0x03)
- **TLS_AES_128_CCM_SHA256**: (0x13,0x04)
- **TLS_AES_128_CCM_8_SHA256**: (0x13,0x05)

The corresponding AEAD algorithms AEAD_AES_128_GCM, AEAD_AES_256_GCM, and AEAD_AES_128_CCM are defined in [RFC5116]. AEAD_CHACHA20_POLY1305 is defined in [RFC7539]. AEAD_AES_128_CCM_8 is defined in [RFC6655]. The corresponding hash algorithms are defined in [SHS].

**Starting point:** agreement on keys & ciphersuite

Similar crypto construction (Wegman-Carter-Shoup)
CHACHA20_POLY1305 for the TLS record

One-time Pad for the MAC

AEAD Key

32 bytes

32 bytes

IV || 0

16 bytes

PRF

64 bytes

IV || 1

16 bytes

PRF

64 bytes

IV || n

16 bytes

PRF

64 bytes

Plaintext (block 1)

64 bytes

Plaintext (block n)

≤ 64 bytes

Additional Data (block 1)

16 bytes

... Additional Data (block n)

≤ 16 bytes

Ciphertext (block 1)

4x16 bytes

... Ciphertext (block n)

≤ 4x16 bytes

lengths of plaintext and additional data

Ciphertext (tag)

16 bytes

16 bytes

Authenticator key

16 bytes
Stream Encryption: Assumptions

One-Time MACs (INT-CMA1)

<table>
<thead>
<tr>
<th>Game UF-1CMA(\mathcal{A}, \text{MAC})</th>
<th>\text{Oracle Mac} (m)</th>
</tr>
</thead>
</table>
| \begin{align*}
  k & \leftarrow \text{MAC.keygen}(\varepsilon); \quad \log \leftarrow \bot \\
  (m^*, t^*) & \leftarrow \mathcal{A}^{\text{Mac}} \\
  \text{return } \text{MAC.verify}(k, m^*, t^*) \\
  \wedge \log \neq (m^*, t^*)
\end{align*} | \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*} |

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

Ciphers (IND-PRF)

<table>
<thead>
<tr>
<th>Game Prf\textsuperscript{b}(PRF)</th>
<th>\text{Oracle Eval} (m)</th>
</tr>
</thead>
</table>
| \begin{align*}
  T & \leftarrow \emptyset \\
  k & \leftarrow \text{PRF.keygen()} \\
  \text{return } \{\text{Eval}\}
\end{align*} | \begin{align*}
  \text{if } T[m] = \bot & \text{ if } b \text{ then } T[m] \leftarrow \text{byte}\textsuperscript{b} \\
  & \text{ else } T[m] \leftarrow \text{PRF.eval}(k, m) \\
  \text{return } T[m]
\end{align*} |

Assumed for AES and Chacha20
Stream Encryption: Assumptions

One-Time MACs (INT-CMA1)

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)

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

many kinds of proofs not just code safety!

arithmetic correctness (field computations)

abstraction & agility

security idealization

injectivity

loops & stateful invariants (reasoning on ideal logs)

TLS-specific mechanisms
- fragmentation
- content multiplexing
- length-hiding, padding
- re-keying
- 0-RTT, 0.5-RTT
# 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>$\epsilon_{Lhse}(A[q_e, q_d]) \leq$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>General bound</strong></td>
<td>$\epsilon_{Prf}(B[q_e (1 + [(2^{14} + 1)/\ell_b])] + q_d + j_0) + \epsilon_{MMac1}(C[2^{14} + 1 + 46, q_d, q_e + q_d])$</td>
</tr>
<tr>
<td>ChaCha20-Poly1305</td>
<td>$\epsilon_{Prf}(B[q_e \left( 1 + \left\lceil \frac{(2^{14}+1)}{64} \right\rceil \right) + q_d \right) + \frac{q_d}{2^{93}}$</td>
</tr>
</tbody>
</table>
| AES128-GCM AES256-GCM| $\epsilon_{Prp}(B[q_b]) + \frac{q_b^2}{2^{129}} + \frac{q_d}{2^{118}}$  
where $q_b = q_e \left( 1 + \left\lceil \frac{(2^{14} + 1)/16} \right\rceil \right) + q_d + 1$ |
| AES128-GCM AES128-GCM| $\frac{q_e}{2^{24.5}} \left( \epsilon_{Prp}(B[2^{34.5}]) \right) + \frac{1}{2^{60}} + \frac{1}{2^{56}}$  
with re-keying every $2^{24.5}$ records (counting $q_b$ for all streams, and $q_d \leq 2^{60}$ per stream) |

$q_e$ 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

$\epsilon_{Lhse}(A[q_e, q_d]) = \epsilon_{Prf} + \epsilon_{MMac1}$

**F* type-based verification on code formalizing game-based reduction**
We verified concrete security on low-level, standard-compliant code (not just a crypto proof on paper)

• Interop as client and server with 3 other implementations of TLS 1.2 and 1.3
• Reasonable performance.

Cost of encrypting a random 2\(^{14}\) fragment

<table>
<thead>
<tr>
<th></th>
<th>Crypto.AEAD</th>
<th>OpenSSL</th>
</tr>
</thead>
<tbody>
<tr>
<td>ChaCha20-Poly1305</td>
<td>13.67 cycles/byte</td>
<td>9.79 cycles/byte</td>
</tr>
<tr>
<td>AES256-GCM</td>
<td>584.80 cycles/byte</td>
<td>33.09 cycles/byte</td>
</tr>
<tr>
<td>AES128-GCM</td>
<td>477.93 cycles/byte</td>
<td>28.27 cycles/byte</td>
</tr>
</tbody>
</table>
Stream Encryption: Performance

Throughput for downloading 1GB of data form a local TLS server

<table>
<thead>
<tr>
<th></th>
<th>OCaml (KB/s)</th>
<th>C (MB/s)</th>
<th>OpenSSL (MB/s)</th>
<th>curl (MB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ChaCha20-Poly1305</td>
<td>167</td>
<td>183</td>
<td>354</td>
<td>440</td>
</tr>
<tr>
<td>AES256-GCM</td>
<td>68</td>
<td>5.61</td>
<td>398</td>
<td>515</td>
</tr>
<tr>
<td>AES128-GCM</td>
<td>89</td>
<td>5.35</td>
<td>406</td>
<td>571</td>
</tr>
</tbody>
</table>

Cost of encrypting a random $2^{14}$ fragment

<table>
<thead>
<tr>
<th></th>
<th>Crypto.AEAD (cycles/byte)</th>
<th>OpenSSL (cycles/byte)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ChaCha20-Poly1305</td>
<td>13.67</td>
<td>9.79</td>
</tr>
<tr>
<td>AES256-GCM</td>
<td>584.80</td>
<td>33.09</td>
</tr>
<tr>
<td>AES128-GCM</td>
<td>477.93</td>
<td>28.27</td>
</tr>
<tr>
<td>Module Name</td>
<td>Verification Goals</td>
<td>LoC</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-------------------------------------------------------------------------------------</td>
<td>-----</td>
</tr>
<tr>
<td>StreamAE</td>
<td>Game StAE(^b) from §VI</td>
<td>318</td>
</tr>
<tr>
<td>AEADProvider</td>
<td>Safety and AEAD security (high-level interface)</td>
<td>412</td>
</tr>
<tr>
<td>Crypto.AEAD</td>
<td>Proof of Theorem 2 from §V</td>
<td>5,253</td>
</tr>
<tr>
<td>Crypto.Plain</td>
<td>Plaintext module for AEAD</td>
<td>133</td>
</tr>
<tr>
<td>Crypto.AEAD.Encoding</td>
<td>AEAD encode function from §V and injectivity proof</td>
<td>478</td>
</tr>
<tr>
<td>Crypto.Symmetric.PRF</td>
<td>Game PrfCt(^b) from §IV</td>
<td>587</td>
</tr>
<tr>
<td>Crypto.Symmetric.Cipher</td>
<td>Agile PRF functionality</td>
<td>193</td>
</tr>
<tr>
<td>Crypto.Symmetric.AES</td>
<td>Safety and correctness w.r.t pure specification</td>
<td>1,254</td>
</tr>
<tr>
<td>Crypto.Symmetric.Chacha20</td>
<td>Safety and correctness w.r.t pure specification</td>
<td>965</td>
</tr>
<tr>
<td>Crypto.Symmetric.UFICMA</td>
<td>Game MMac(^1) from §III</td>
<td>617</td>
</tr>
<tr>
<td>Crypto.Symmetric.MAC</td>
<td>Agile MAC functionality</td>
<td>488</td>
</tr>
<tr>
<td>Crypto.Symmetric.GF128</td>
<td>(GF(128)) polynomial evaluation and GHASH encoding</td>
<td>306</td>
</tr>
<tr>
<td>Crypto.Symmetric.Poly1305</td>
<td>(GF(2^{130} - 5)) polynomial evaluation and Poly1305 encoding</td>
<td>604</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>
</tr>
<tr>
<td>FStar.Buffer.*</td>
<td>A verified model of mutable buffers (implemented natively)</td>
<td>1,340</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td>15,480</td>
</tr>
</tbody>
</table>
TLS 1.3 Handshake (Outline)
TLS protocol overview

**Hello**
- **Protocol negotiation**
  - Agree on version
  - Agree on ciphersuite
  - Determines all crypto algos

**Keying**
- Authenticated Key Exchange
  - Verify server/client identity
  - Generate master secret
  - Derive connection keys

**Finished**
- Key & transcript confirmation
  - Completes authentication
  - Matches transcripts
  - Authenticated encryption

**AppData**
- Application data streams
  - Full duplex channel
  - Authenticated encryption
Negotiation

Configuration

Certificates

Signing

Syntax: parse/format

Messages

Extensions

Session Log

Flights

Digests

Hash

Key Schedule

State machine

Keys

Shares

HMAC

ODH

PRF

Handshake

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

high-level formatter
val formatCH: clientHello ->
  bytes

low-level serializer
val serializeCH: output: buffer ->
  len: UInt32.t -> pv: ... -> ... ->
  Heap (option UInt32.t) ...
  (ensures fun h0 result h1 ->
    modifies h0 output.[0..len-1] h1 /
    match result with
    | Some pos -> ...
  )
Low-level parsing: variable-length bytes

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

let parse_vlbytes₁ (#t: Type₀) (p: parser t): parser t = parse_u₈ `and_then` (λ len → parse_sized₁ p len)
Diffie-Hellman shared secret ($g^{xy}$)

early secret

handshake secret

master secret

Encryption

Export (QUIC)

Integrity

Handshake Integrity

Export (QUIC)

Integrity

Encryption

Caption: two kinds of key derivation steps

key materials

Extract

prior secret

transcript

label

new secret

prior secret

derived secret

pre-shared keys for future sessions
Our (fresh) crypto model precisely reflects F* code modularity, involves a security definition for each color, supports agility and key compromise.
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://fstarlang.org