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

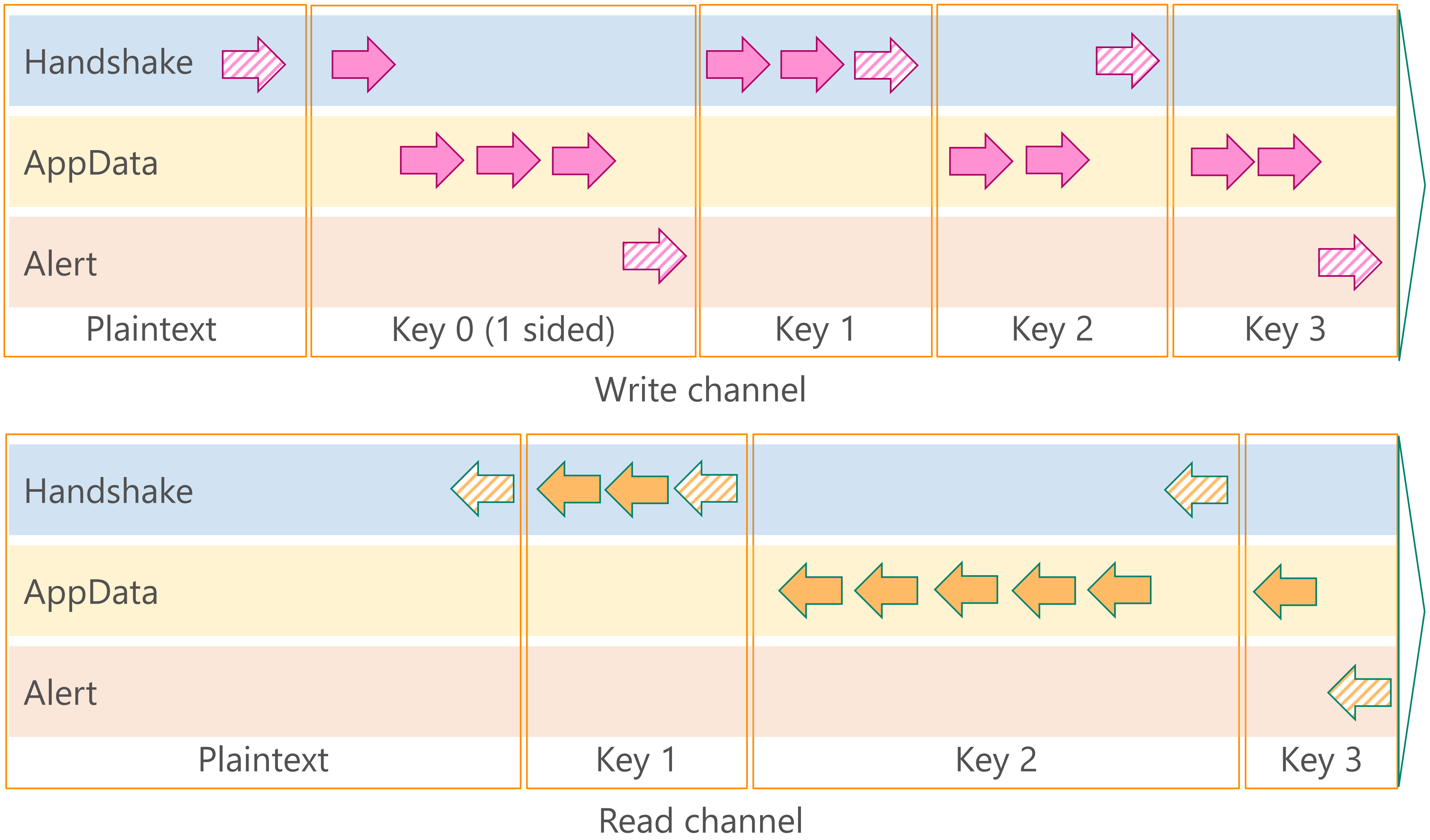
Standardized complications

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

Poor TLS track record

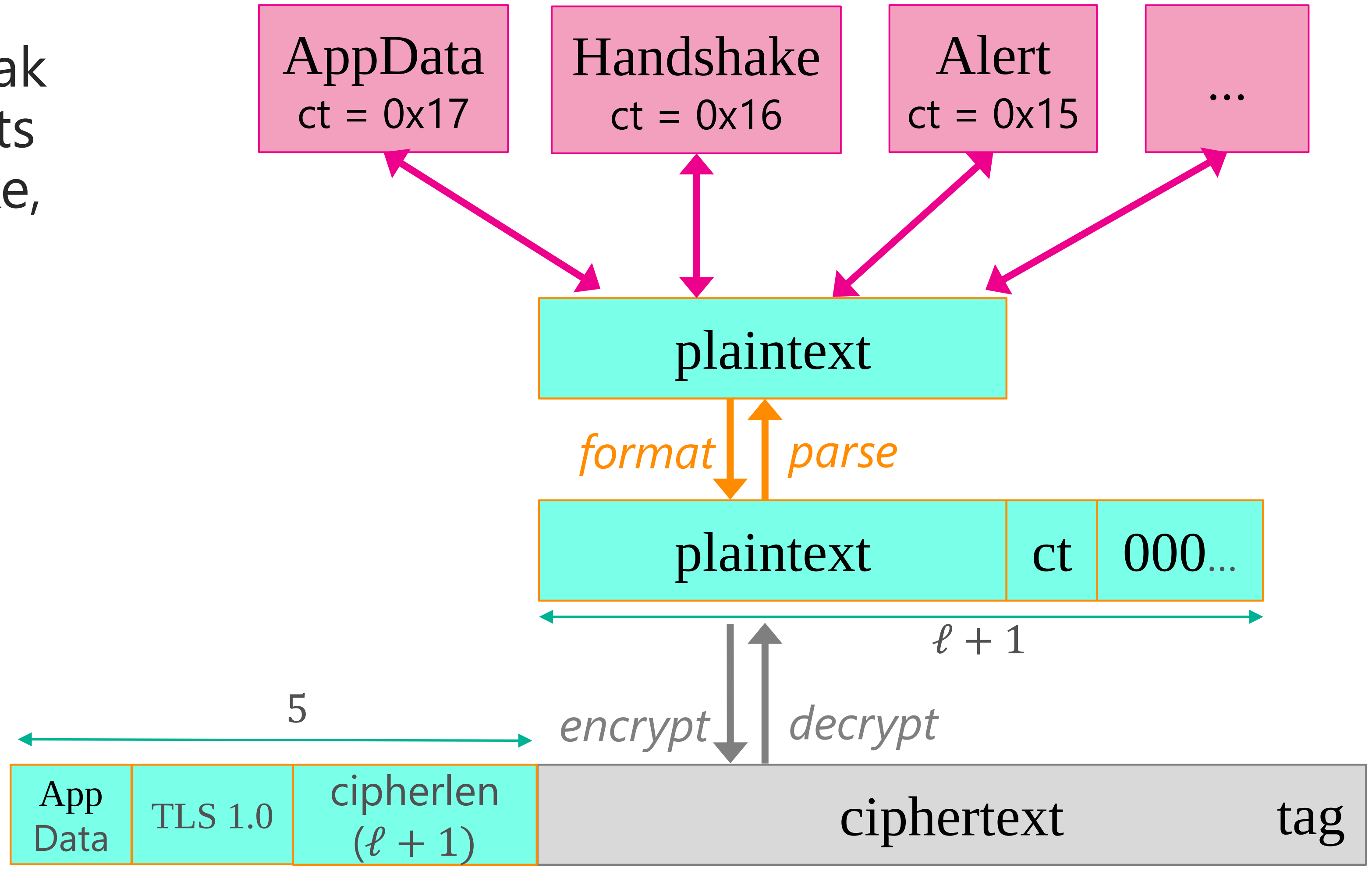
- Many implementation flaws
- Attacks on weak cryptography (MD5, SHA1, …)
- Attacks on weak constructions (MAC-Encode-then-Encrypt)
- Attacks on compression
- Persistent side channels
- Persistent truncation attacks

The TLS Record Layer



The TLS Record Layer (TLS 1.3)

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

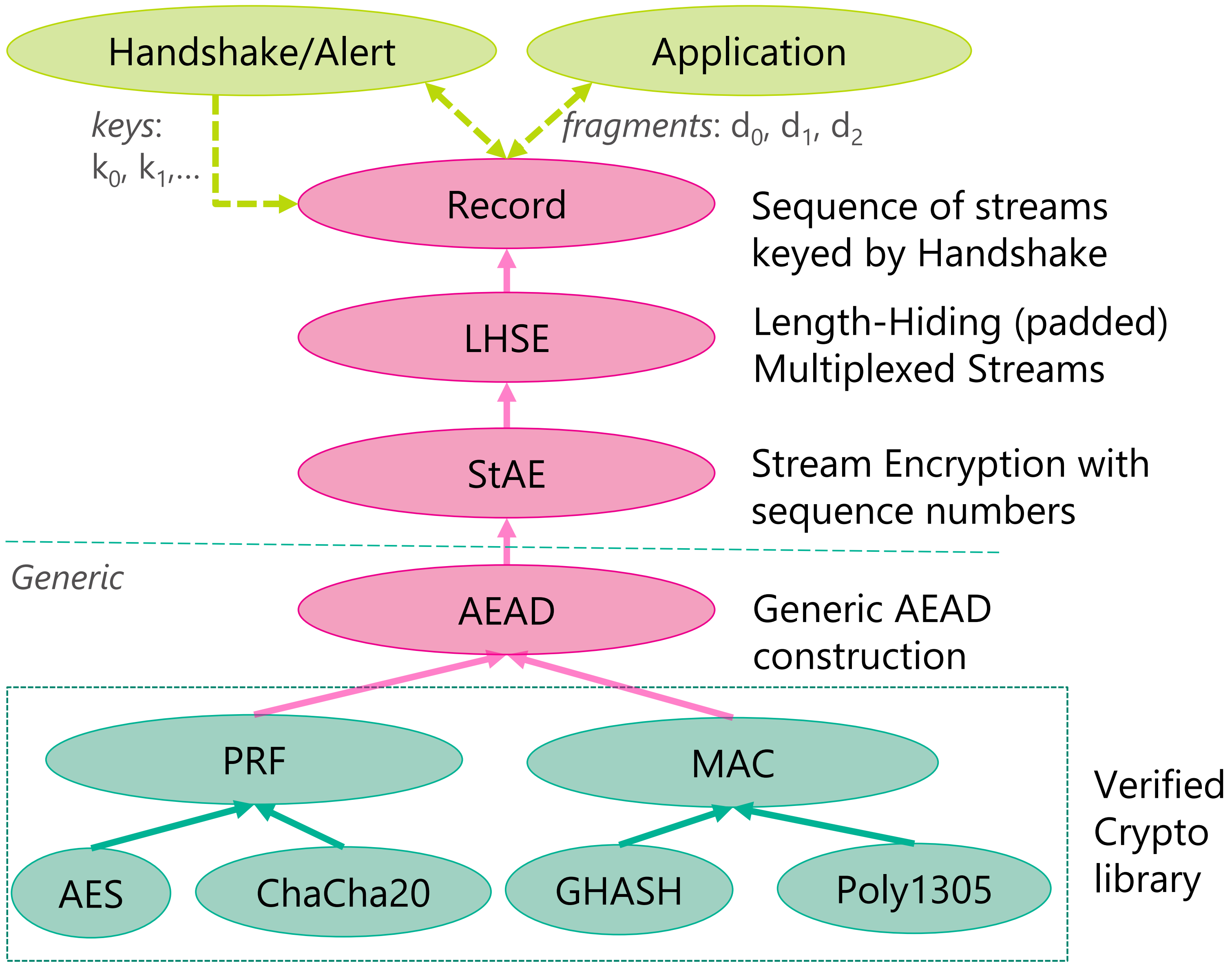


The TLS Record Layer (F*)

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

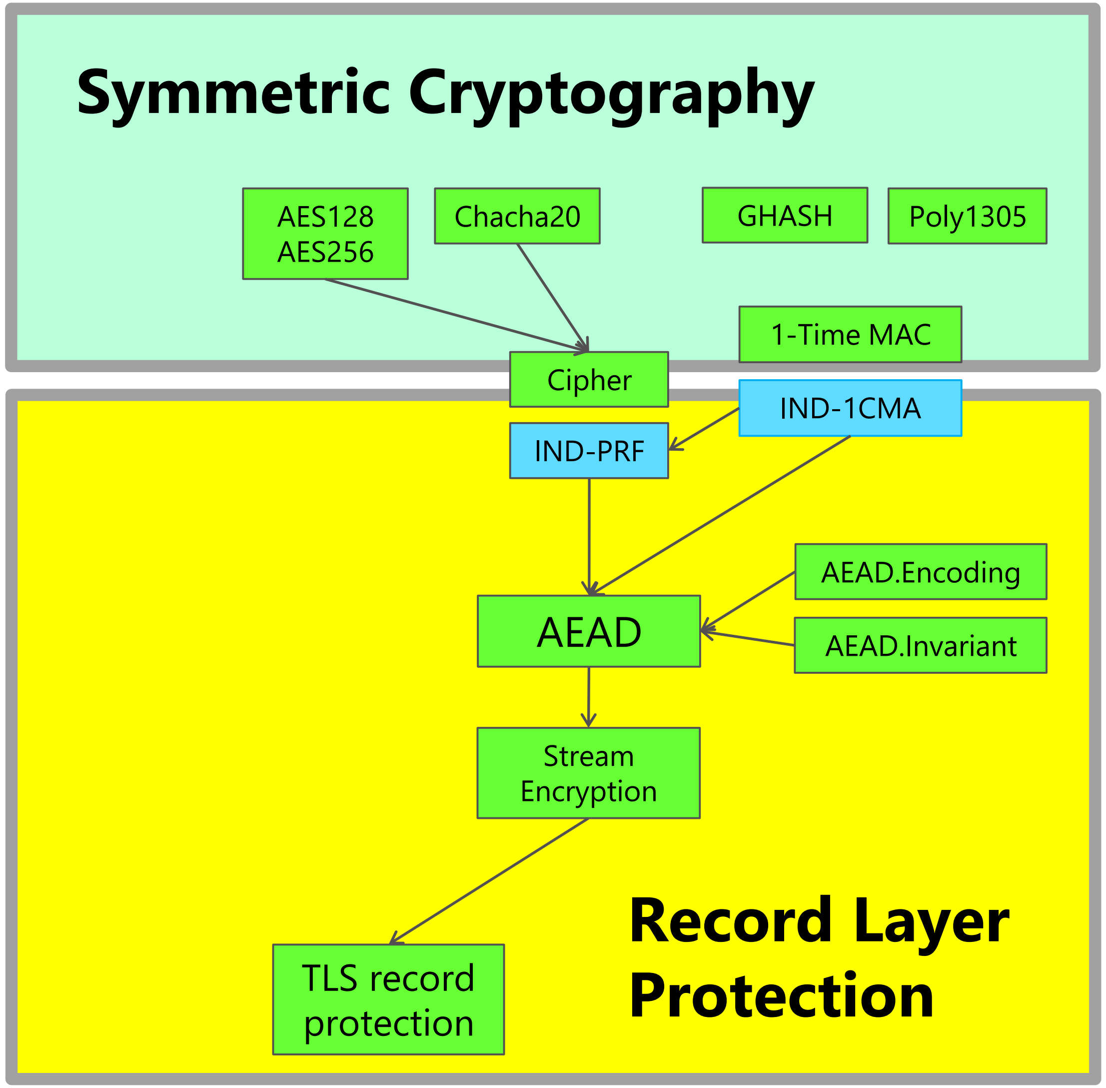
We make code-based security assumptions on the crypto primitives (PRF, MAC)

We obtain security guarantees at the top-level API for the TLS record layer

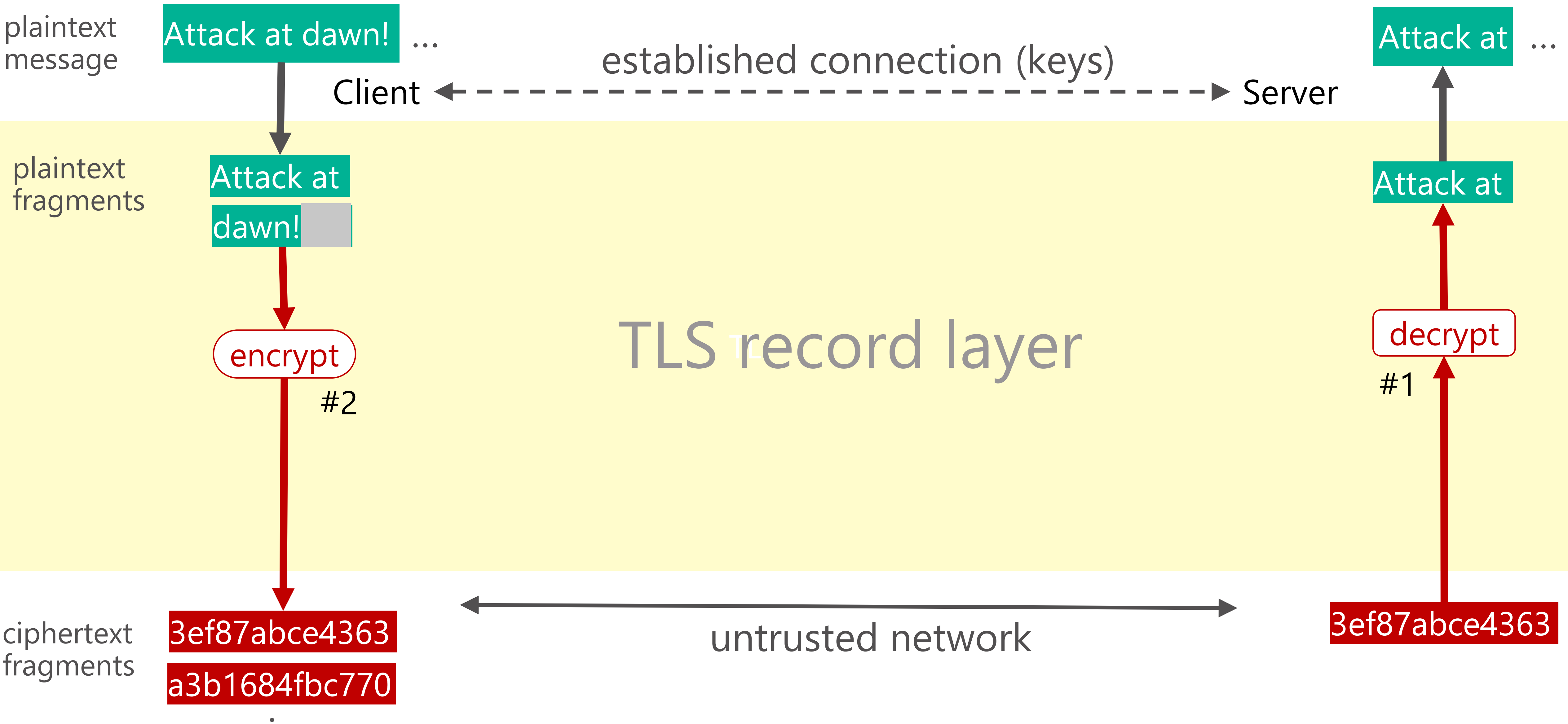


Crypto security for TLS Stream Encryption

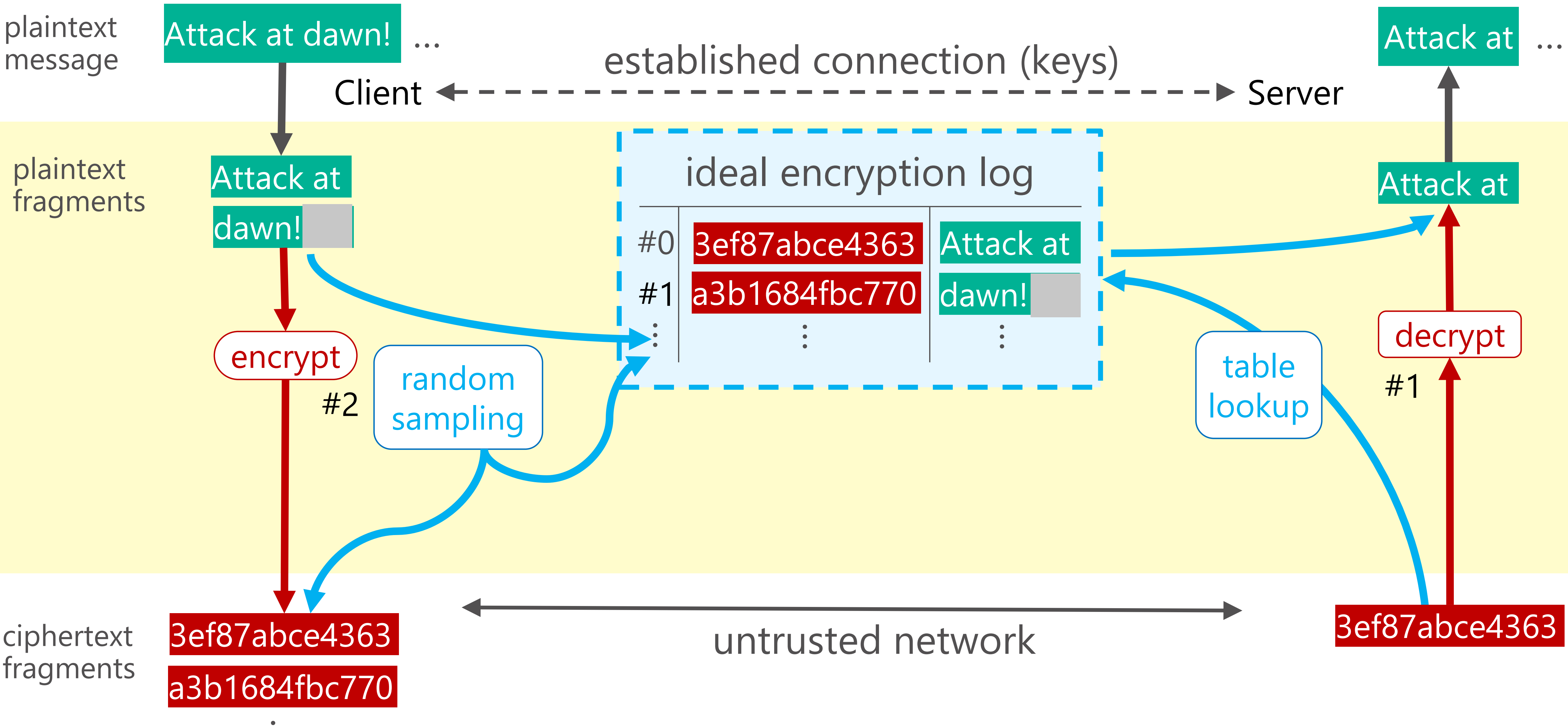
1. Security definition
2. TLS constructions (AEAD)
3. Concrete security bounds
4. Verification
5. Performance



Stream Encryption: Security Definition



Stream Encryption: Security Definition



AEAD Encryption: Security Definition

Game $\mathsf{RoR}(AEAD)$

$\mathsf{log} \leftarrow \varnothing;$
$b \leftarrow\!\!\$\ \{0,1\}$
$k \leftarrow\!\!\$\ \mathsf{keygen}()$
$b' \leftarrow\!\!\$\ \mathcal{A}^{\mathsf{Encrypt},\mathsf{Decrypt}}()$
return $(b' = b)$

Oracle $\mathsf{Encrypt}(n, a, m)$

if $\mathsf{log}[n] \neq \bot$ **return** $\bot$
if b
  $c \leftarrow\!\!\$\ Byte^{|\mathsf{m}|+\mathsf{taglen}}$
else
  $c \leftarrow \mathsf{encrypt}(k, n, a, m)$
$\mathsf{log}[n] \leftarrow (a, m, c)$
return c

Oracle $\mathsf{Decrypt}(n, a, c)$

if $b = 1$
  if $\mathsf{log}[n] = (a, m, c)$ **return** $\mathsf{Some}(m)$
  else return None
else return $\mathsf{decrypt}(k, n, a, c)$

Stream encryption in TLS 1.3

Starting point: agreement on keys & ciphersuite

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)

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;
```

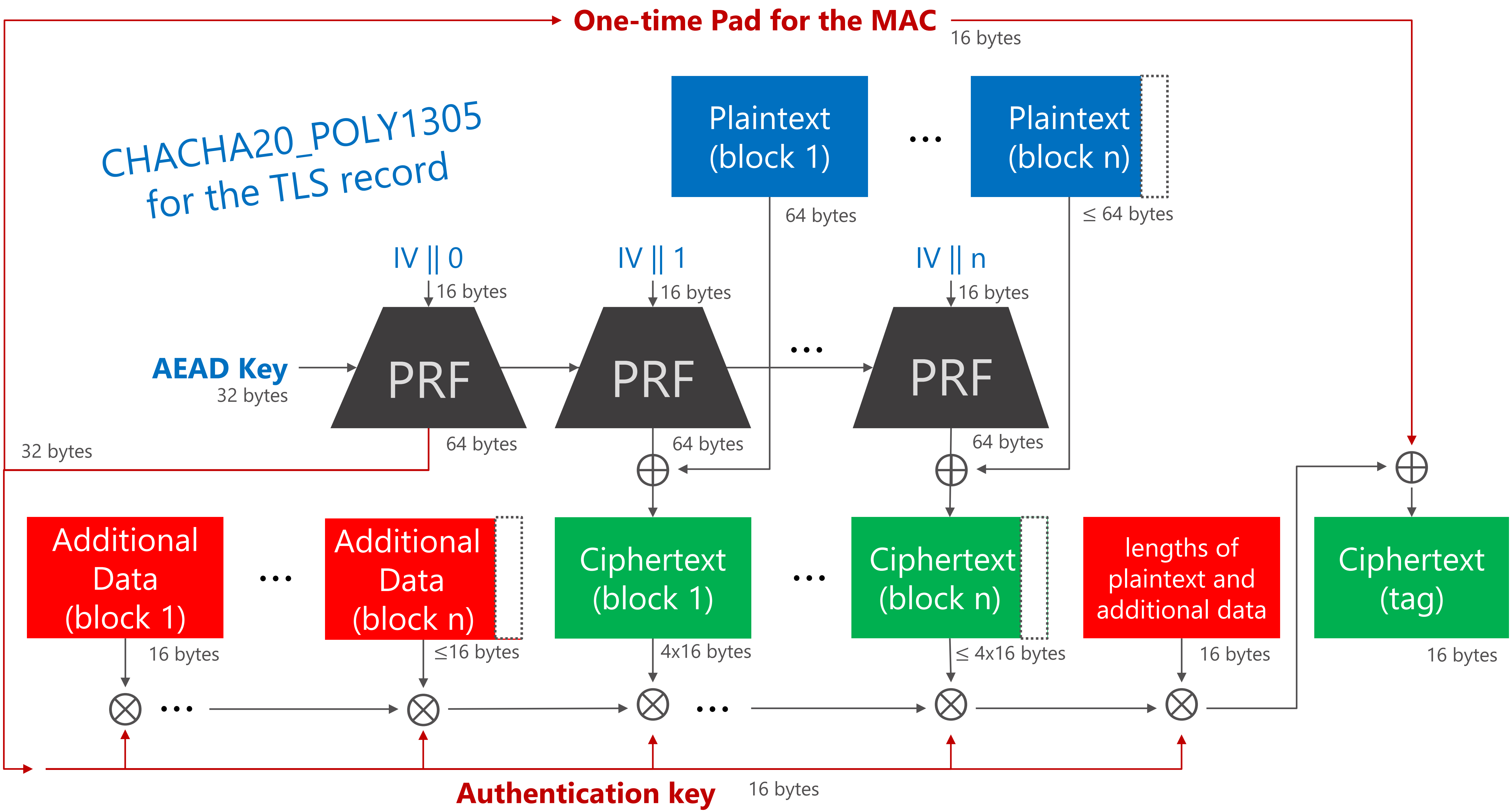
Component	Contents
TLS	The string "TLS"
AEAD	The AEAD algorithm used for record protection
HASH	The hash algorithm used with HKDF
VALUE	The two byte ID assigned for this cipher suite

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

Description	Value
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}

Similar crypto construction (Wegman-Carter-Shoup)

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].



One-time Pad for the MAC
16 bytes
CHACHA20_POLY1305
for the TLS record
Plaintext (block 1)
64 bytes
Plaintext (block n)
≤ 64 bytes
IV || 0
16 bytes
IV || 1
16 bytes
IV || n
16 bytes
AEAD Key
32 bytes
PRF
PRF
PRF
32 bytes
64 bytes
64 bytes
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
16 bytes
Ciphertext (tag)
16 bytes
Authentication key
16 bytes

Stream Encryption: Assumptions

One-Time MACs (INT-CMA1)

Game UF-1CMA$(\mathcal{A}, \mathsf{MAC})$

$k \xleftarrow{\$} \mathsf{MAC.keygen}(\varepsilon);\ \mathit{log} \leftarrow \bot$
$(m^\star, t^\star) \leftarrow \mathcal{A}^{\mathsf{Mac}}$
return $\mathsf{MAC.verify}(k, m^\star, t^\star)$
$\quad \wedge\ \mathit{log} \neq (m^\star, t^\star)$

Oracle Mac(m)

if $\mathit{log} \neq \bot$ **return** $\bot$
$t \leftarrow \mathsf{MAC.mac}(k, m)$
$\mathit{log} \leftarrow (m, t)$
return t

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

Ciphers (IND-PRF)

Game $\mathsf{Prf}^b(\mathsf{PRF})$

$T \leftarrow \varnothing$
$k \xleftarrow{\$} \mathsf{PRF.keygen}()$
return $\{\mathsf{Eval}\}$

Oracle Eval(m)

if $T[m] = \bot$
$\quad$ **if** b **then** $T[m] \xleftarrow{\$} \mathsf{byte}^{\ell_b}$
$\quad$ **else** $T[m] \leftarrow \mathsf{PRF.eval}(k, m)$
return $T[m]$

Assumed for AES and Chacha20

Stream Encryption: Assumptions

One-Time MACs (INT-CMA1)

Game UF-1CMA($\mathcal{A}$, MAC)

$k \xleftarrow{\$} \mathsf{MAC.keygen}(\varepsilon);\ log \leftarrow \bot$
$(m^\star, t^\star) \leftarrow \mathcal{A}^{\mathsf{Mac}}$
return $\mathsf{MAC.verify}(k, m^\star, t^\star)$
$\quad \wedge\ log \neq (m^\star, t^\star)$

Oracle Mac(m)

if $log \neq \bot$ **return** $\bot$
$t \leftarrow \mathsf{MAC.mac}(k, m)$
$log \leftarrow (m, t)$
return t

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)

Game Prf^b(PRF)

$T \leftarrow \varnothing$
$k \xleftarrow{\$} \mathsf{PRF.keygen}()$
return {Eval}

Oracle Eval(m)

if $T[m] = \bot$
$\quad$ **if** b **then** $T[m] \xleftarrow{\$} \mathsf{byte}^{\ell_b}$
$\quad$ **else** $T[m] \leftarrow \mathsf{PRF.eval}(k, m)$
return $T[m]$

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

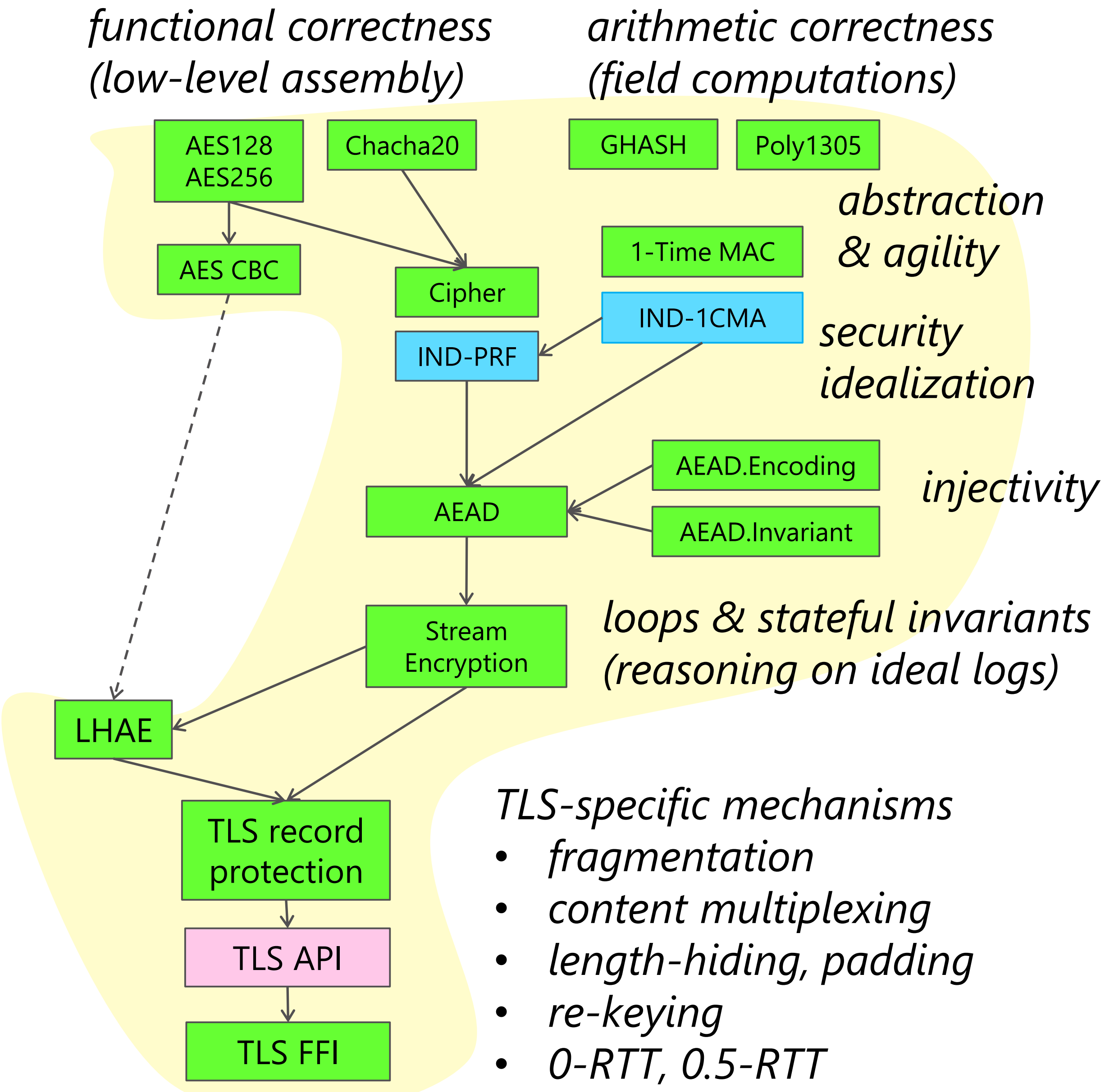
Stream Encryption: Construction

many kinds of proofs
not just code safety!

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



Stream Encryption: Concrete Bounds

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

Ciphersuite	$\epsilon_{\mathsf{Lhse}}(\mathcal{A}[q_e, q_d]) \le$
General bound	$\epsilon_{\mathsf{Prf}}(\mathcal{B}[q_e(1 + \lceil(2^{14} + 1)/\ell_b\rceil) + q_d + j_0]) + \epsilon_{\mathsf{MMac1}}(\mathcal{C}[2^{14} + 1 + 46, q_d, q_e + q_d])$
ChaCha20-Poly1305	$\epsilon_{\mathsf{Prf}}\left(\mathcal{B}\left[q_e\left(1 + \left\lceil\frac{(2^{14}+1)}{64}\right\rceil\right) + q_d\right]\right) + \frac{q_d}{2^{93}}$
AES128-GCM AES256-GCM	$\epsilon_{\mathsf{Prp}}(\mathcal{B}[q_b]) + \frac{q_b^2}{2^{129}} + \frac{q_d}{2^{118}}$ where $q_b = q_e(1 + \lceil(2^{14} + 1)/16\rceil) + q_d + 1$
AES128-GCM AES128-GCM	$\frac{q_e}{2^{24.5}}\left(\epsilon_{\mathsf{Prp}}(\mathcal{B}[2^{34.5}]) + \frac{1}{2^{60}} + \frac{1}{2^{56}}\right)$ with re-keying every $2^{24.5}$ records (counting q_b for all streams, and $q_d \le 2^{60}$ per stream)

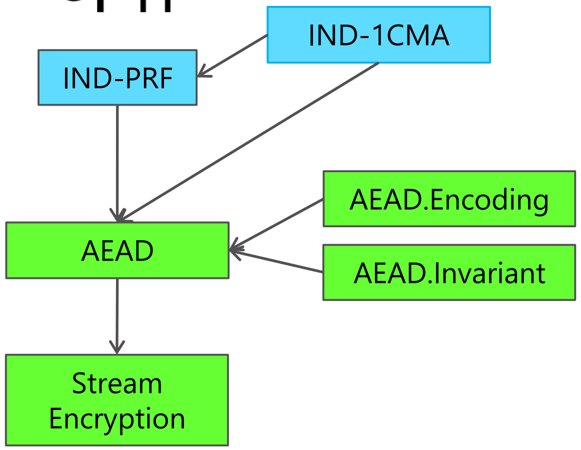
q_e is the number of encrypted records;
q_d is the number of chosen-ciphertext decryptions;
q_b is the total number of blocks for the PRF

Standard crypto assumption

ϵ_{Prf}

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

$$\epsilon_{\mathsf{MMac1}} = \frac{d.\tau.q_v}{|R|}$$



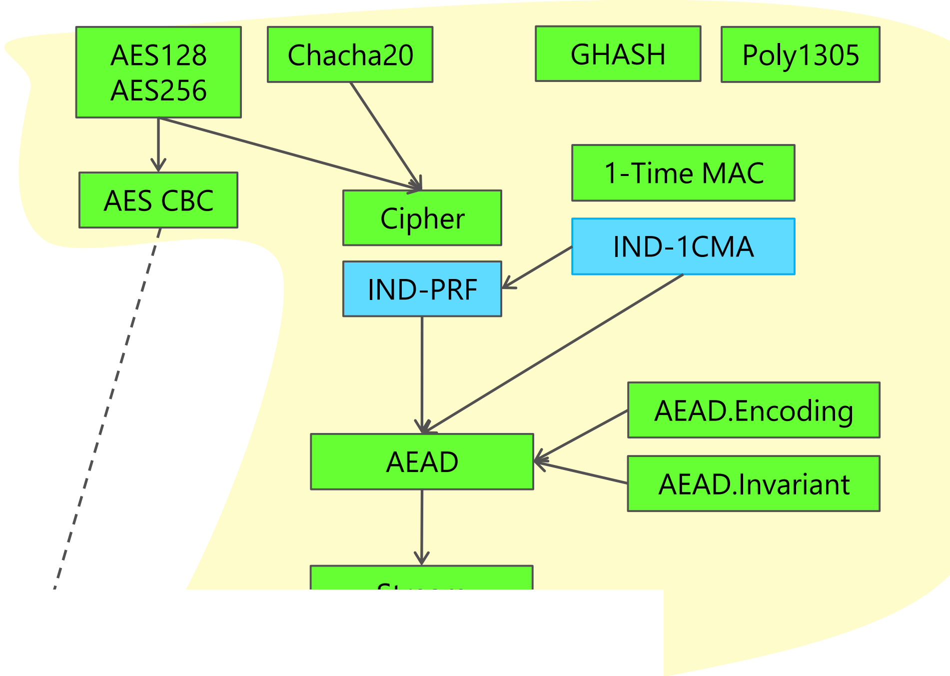
$$\epsilon_{\mathsf{Lhse}}(\mathcal{A}[q_e, q_d]) = \epsilon_{\mathsf{Prf}} + \epsilon_{\mathsf{MMac1}}$$

F type-based verification on code formalizing game-based reduction*

Stream Encryption: Performance

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

	Crypto.AEAD	OpenSSL
ChaCha20-Poly1305	13.67 cycles/byte	9.79 cycles/byte
AES256-GCM	584.80 cycles/byte	33.09 cycles/byte
AES128-GCM	477.93 cycles/byte	28.27 cycles/byte

Stream Encryption: Performance

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

	OCaml	C	OpenSSL	curl
ChaCha20-Poly1305	167 KB/s	183 MB/s	354 MB/s	440 MB/s
AES256-GCM	68 KB/s	5.61 MB/s	398 MB/s	515 MB/s
AES128-GCM	89 KB/s	5.35 MB/s	406 MB/s	571 MB/s

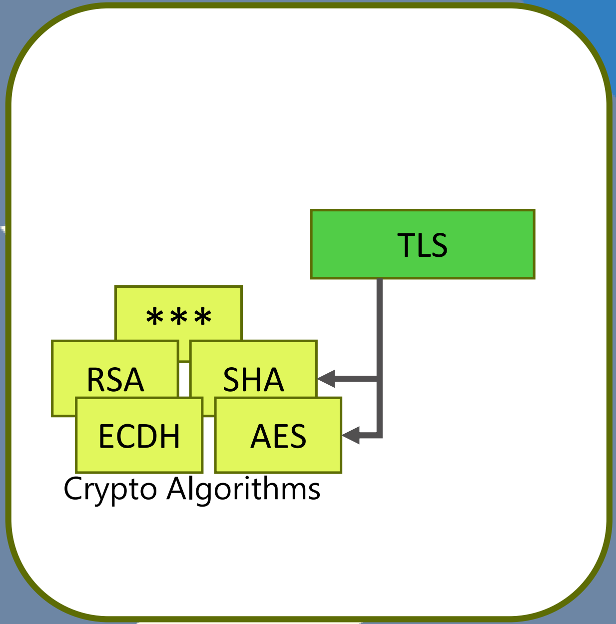
Cost of encrypting a random 2^14 fragment

	Crypto.AEAD	OpenSSL
ChaCha20-Poly1305	13.67 cycles/byte	9.79 cycles/byte
AES256-GCM	584.80 cycles/byte	33.09 cycles/byte
AES128-GCM	477.93 cycles/byte	28.27 cycles/byte

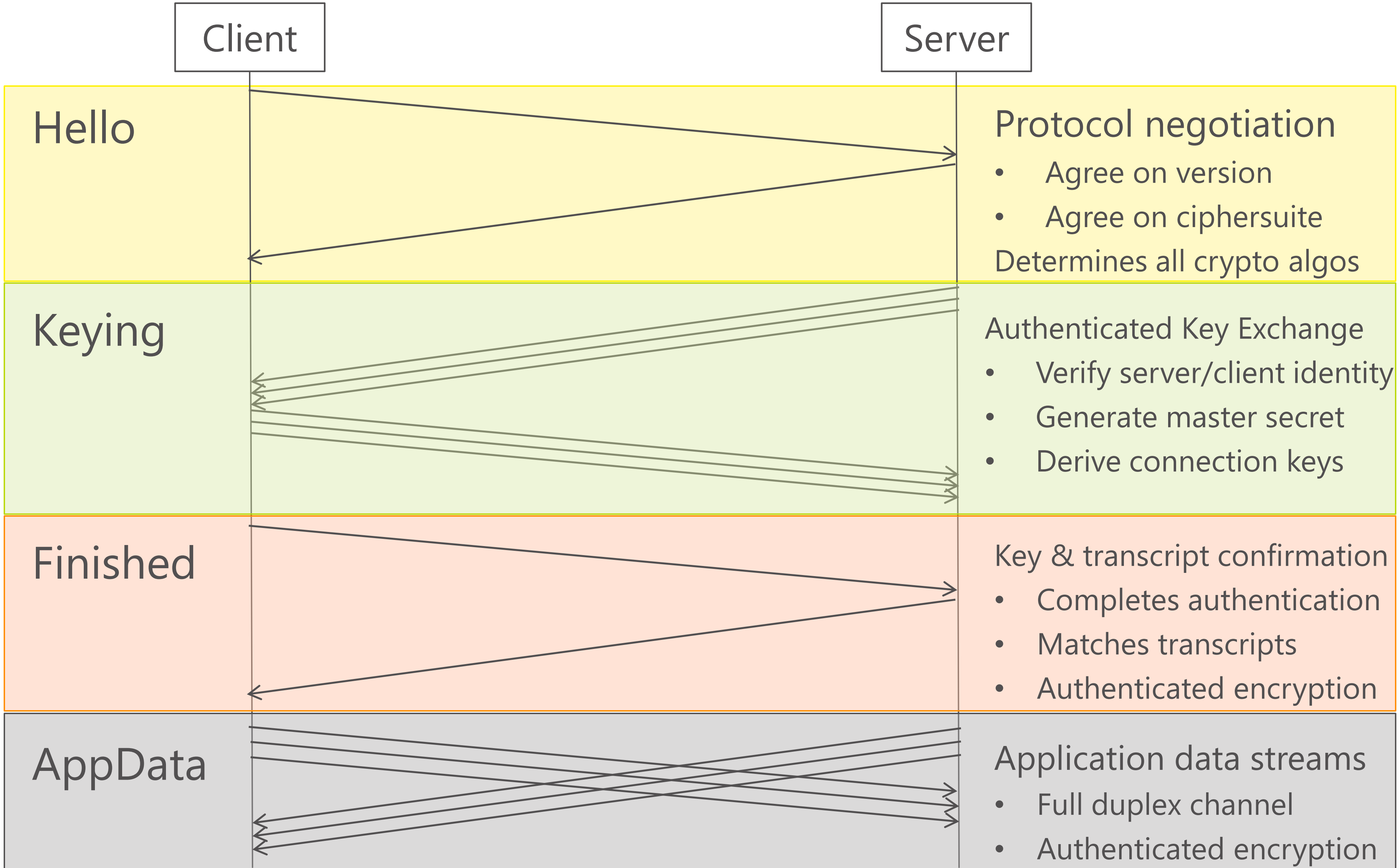
Stream Encryption: Verification Effort

Module Name	Verification Goals	LoC	% annot	ML LoC	C LoC	Time
StreamAE	Game StAE^b from §VI	318	40%	354	N/A	307s
AEADProvider	Safety and AEAD security (high-level interface)	412	30%	497	N/A	349s
Crypto.AEAD	Proof of Theorem 2 from §V	5,253	90%	2,738	2,373	1,474s
Crypto.Plain	Plaintext module for AEAD	133	40%	95	85	8s
Crypto.AEAD.Encoding	AEAD encode function from §V and injectivity proof	478	60%	280	149	708s
Crypto.Symmetric.PRF	Game PrfCtr^b from §IV	587	40%	522	767	74s
Crypto.Symmetric.Cipher	Agile PRF functionality	193	30%	237	270	65s
Crypto.Symmetric.AES	Safety and correctness w.r.t pure specification	1,254	30%	4,672	3,379	134s
Crypto.Symmetric.Chacha20		965	80%	296	119	826s
Crypto.Symmetric.UF1CMA	Game $\mathsf{MMac1}^b$ from §III	617	60%	277	467	428s
Crypto.Symmetric.MAC	Agile MAC functionality	488	50%	239	399	387s
Crypto.Symmetric.GF128	$GF(128)$ polynomial evaluation and GHASH encoding	306	40%	335	138	85s
Crypto.Symmetric.Poly1305	$GF(2^{130}-5)$ polynomial evaluation and Poly1305 encoding	604	70%	231	110	245s
Hacl.Bignum	Bignum library and supporting lemmas for the functional correctness of field operations	3,136	90%	1,310	529	425s
FStar.Buffer.*	A verified model of mutable buffers (implemented natively)	1,340	100%	N/A	N/A	563s
Total		15,480	78%	12,083	8,795	1h 41m

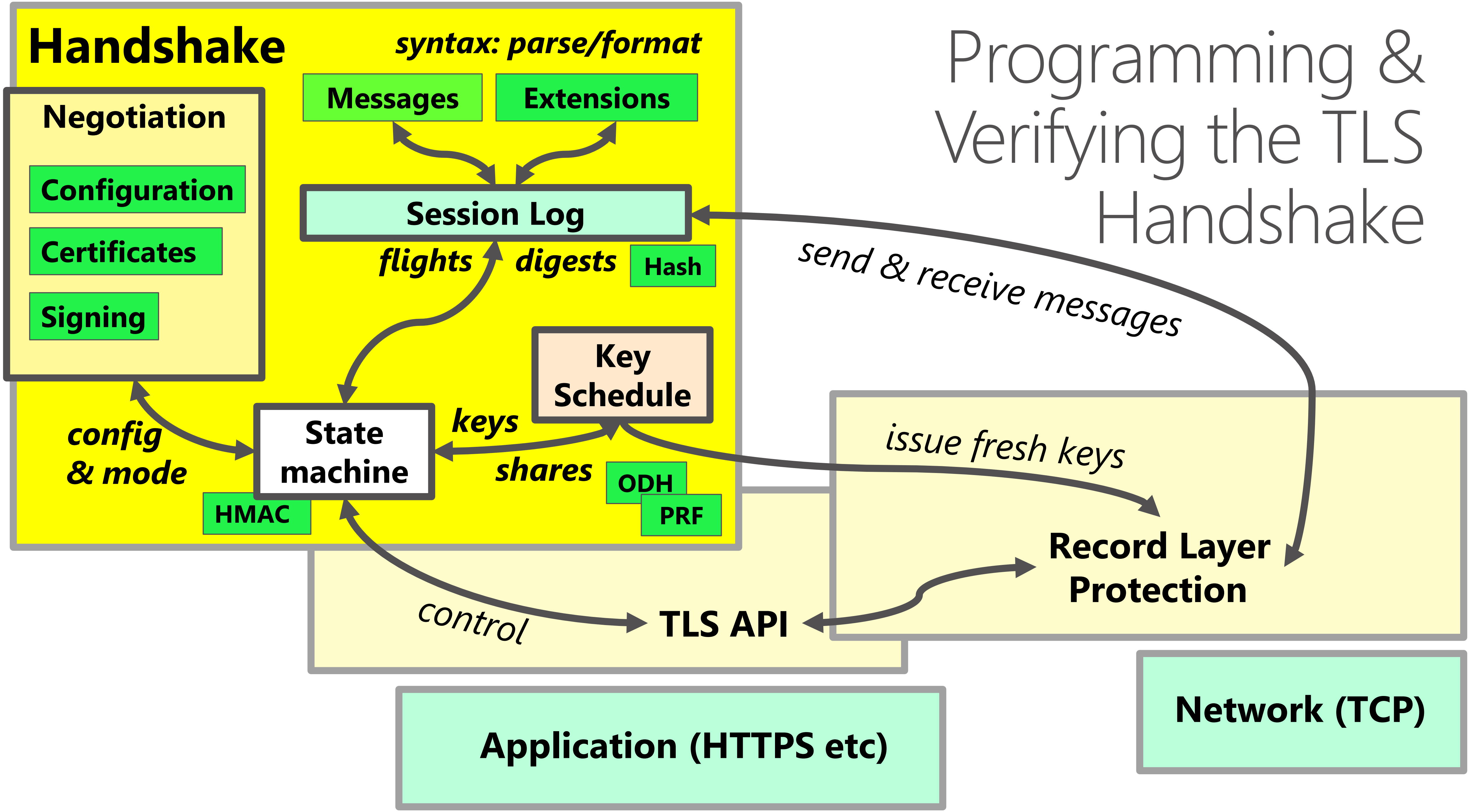
TLS 1.3 Handshake (Outline)



TLS protocol overview



Programming & Verifying the TLS Handshake



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
```

high-level type

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

high-level formatter

```
val formatCH:
  clientHello ->
  bytes
```

inverse properties

```
val injCH:
  clientHello ->
  Lemma …
```

```
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;
```

erased specification

---

low-level in-place code extracted to C

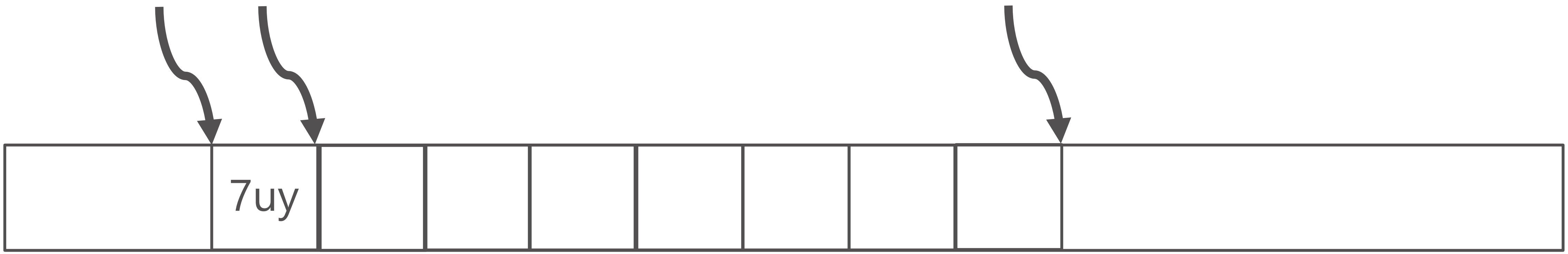
low-level validator

```
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)
```

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 -> … //idem
```

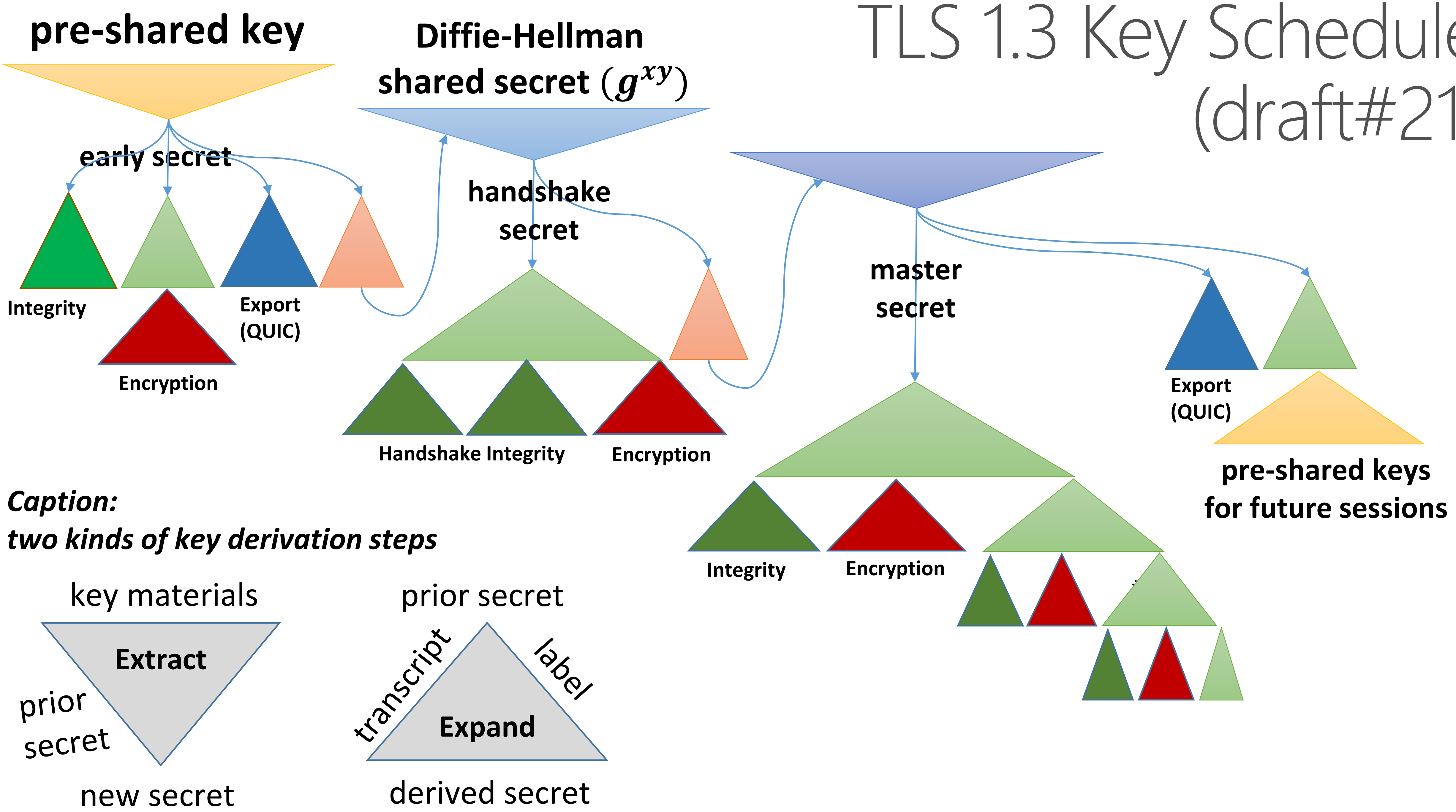
Low-level parsing: variable-length bytes



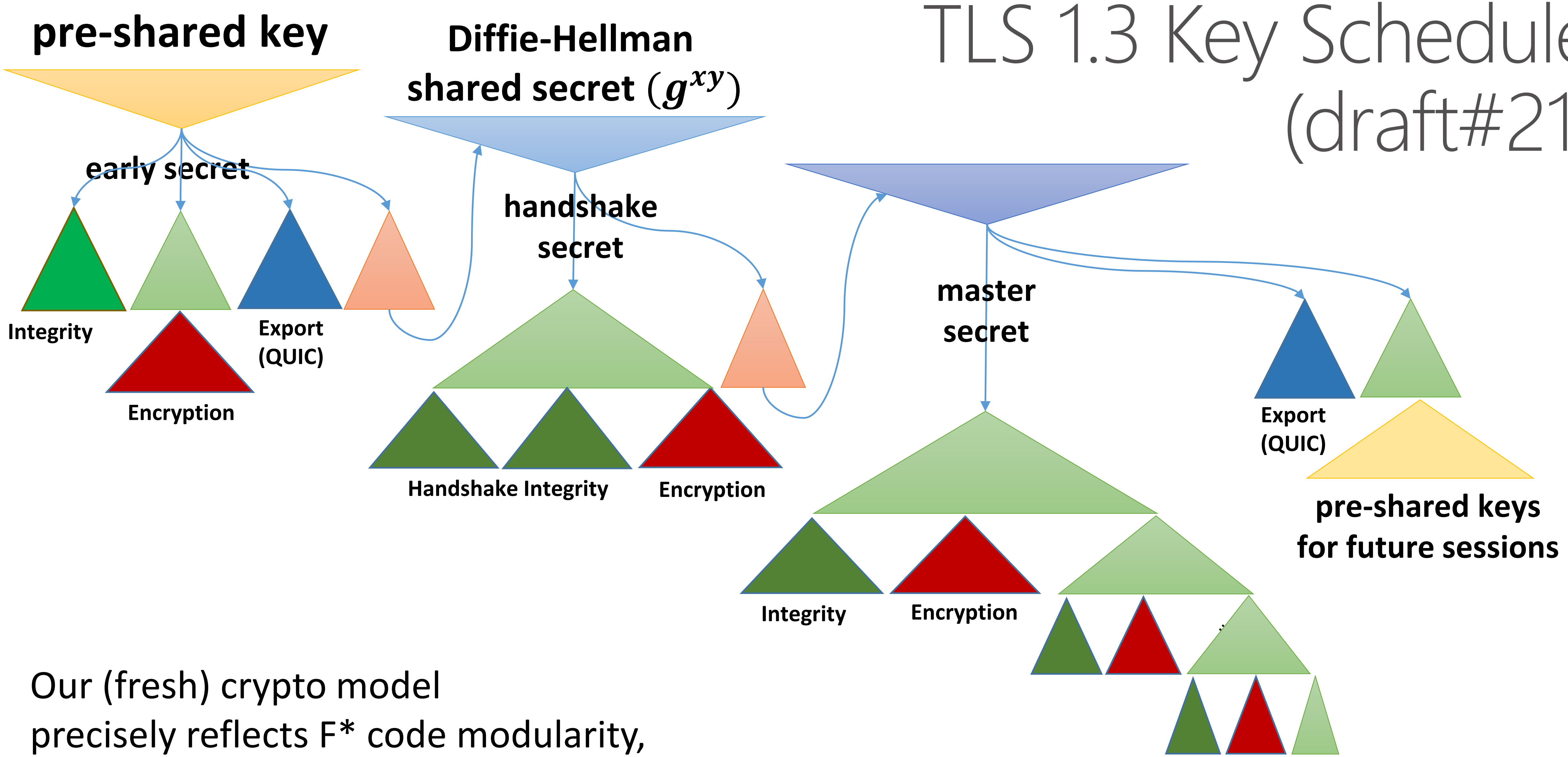
e.g. session_id <0..32> is formatted as a “vlbytes 1”

```
let parse_vlbytes₁ (#t: Type₀) (p: parser t): parser t =
  parse_u₈ `and_then` (λ len → parse_sized₁ p len)
```

TLS 1.3 Key Schedule (draft#21)



TLS 1.3 Key Schedule (draft#21)



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