Softtware Verification Spring School, Aussois, May 7—11 2018

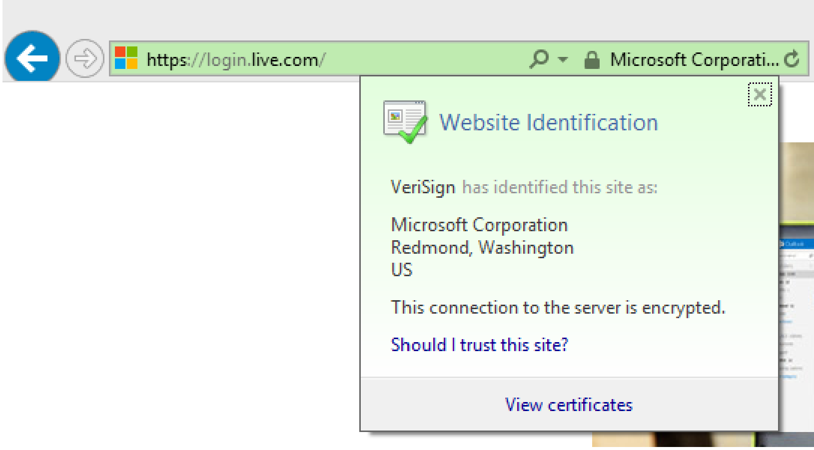
Security Verification and cryptographic modelling in F*

Antoine Delignat-Lavaud
Catalin Hritcu
Danel Ahman

Microsoft Research

Inria INVENTEURS DU MONDE NUMÉRIQUE

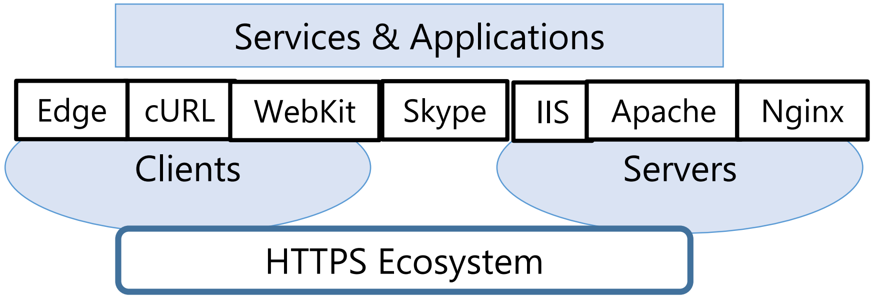
Microsoft Research - Inria JOINT CENTRE



Everest*: Verified Drop-in Replacements for TLS/HTTPS

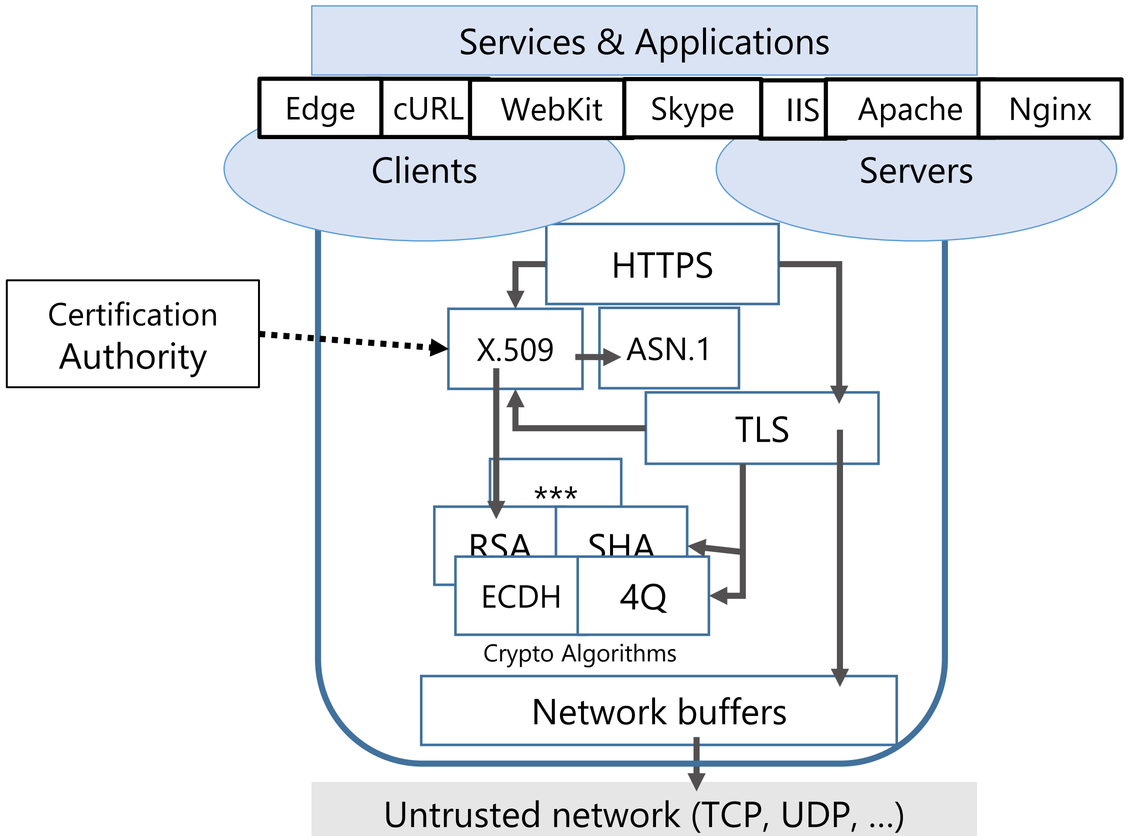
*the Everest VERified End-to-end Secure Transport

The HTTPS Ecosystem is critical



- Default protocol—trillions of connections
- Most of Internet traffic (+40%/year)
- Web, cloud, email, VoIP, 802.1x, VPNs, IoT…

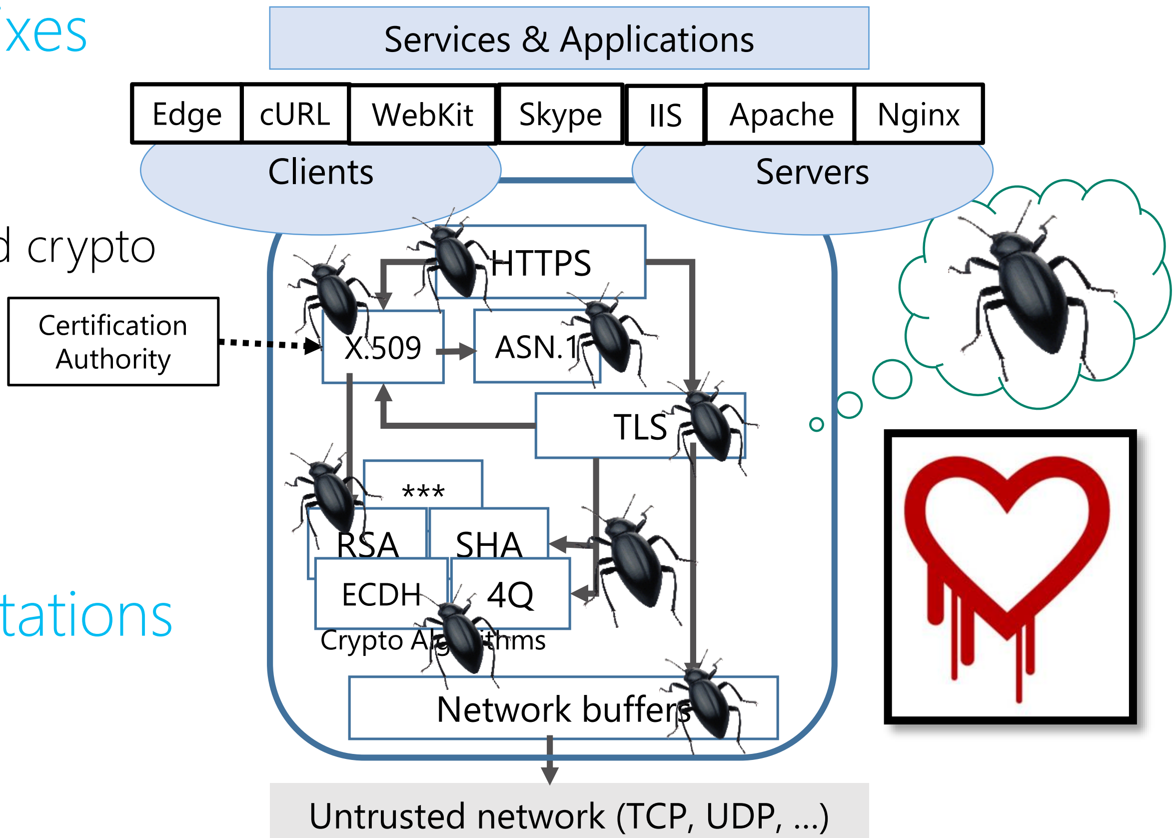
The HTTPS Ecosystem is complex



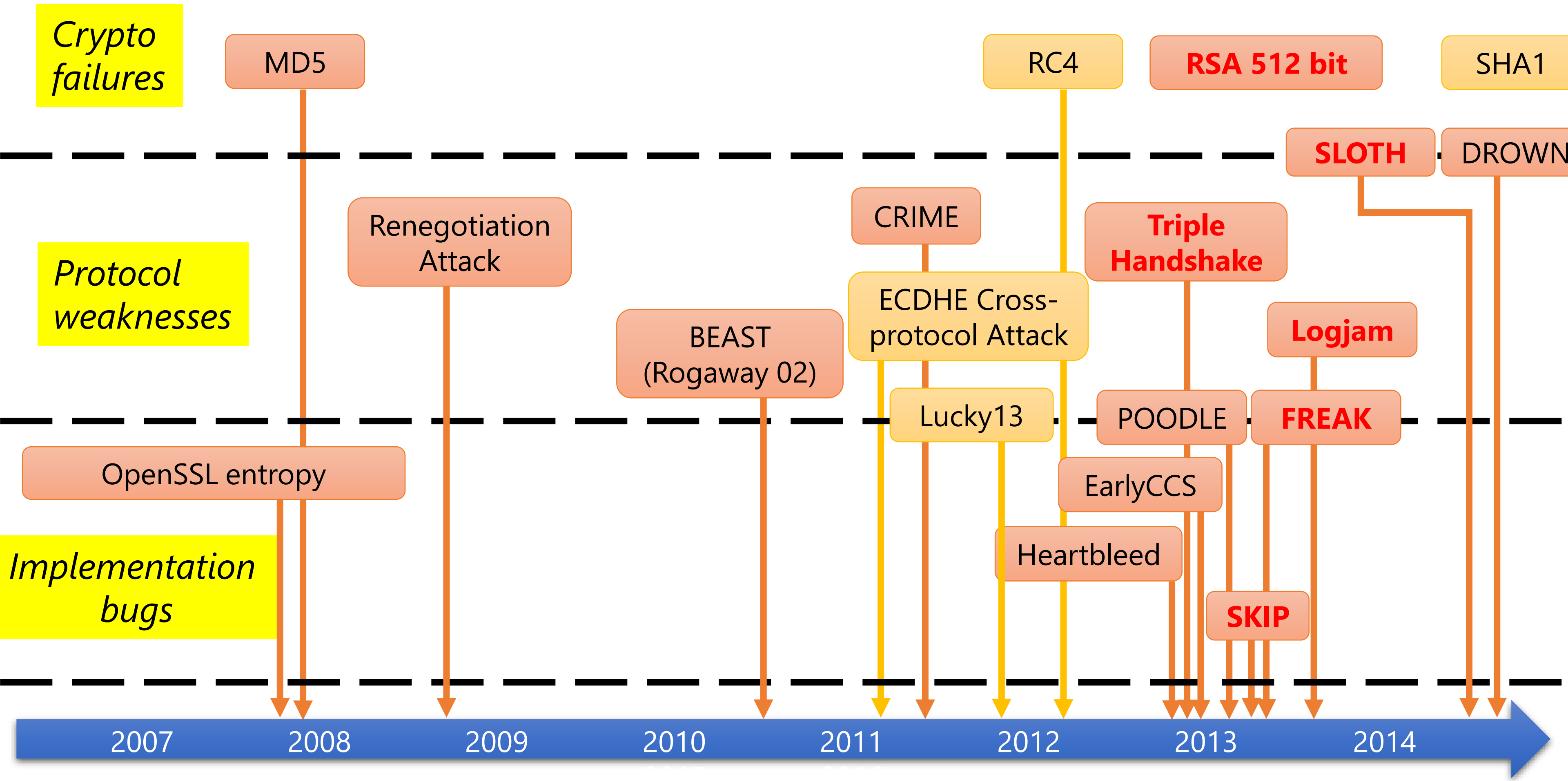
The HTTPS Ecosystem is broken

- 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

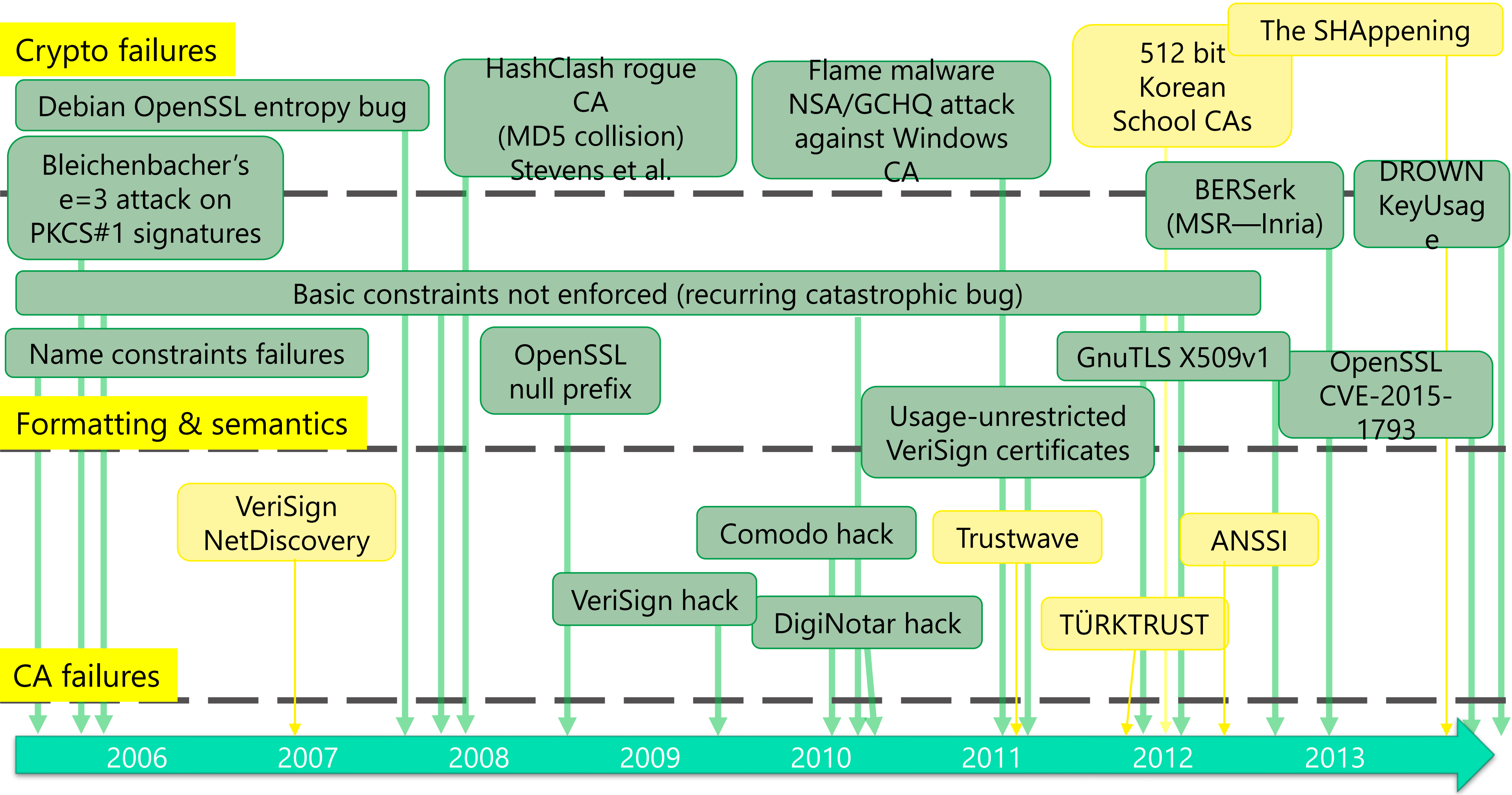
- Mainstream implementations
  - OpenSSL, SChannel, NSS, …
  - Monthly security patches



A Timeline of Recent TLS Attacks

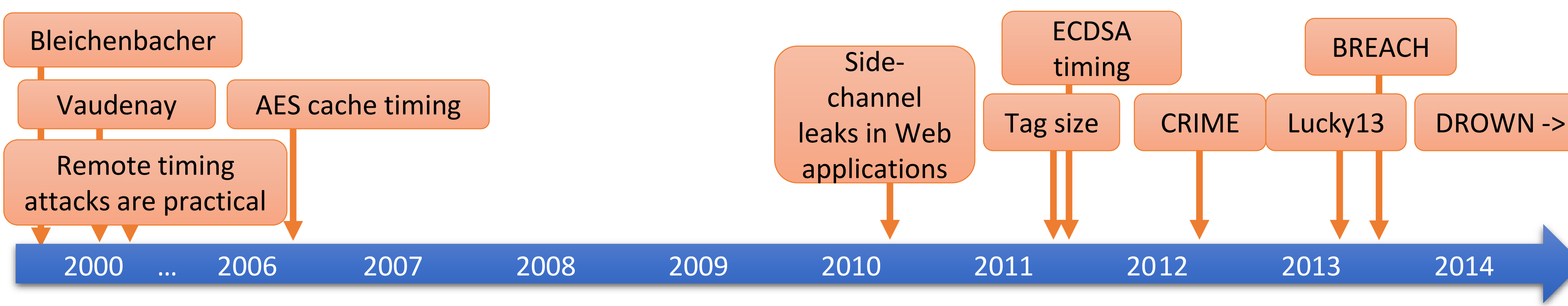


A Timeline of Recent PKI Failures



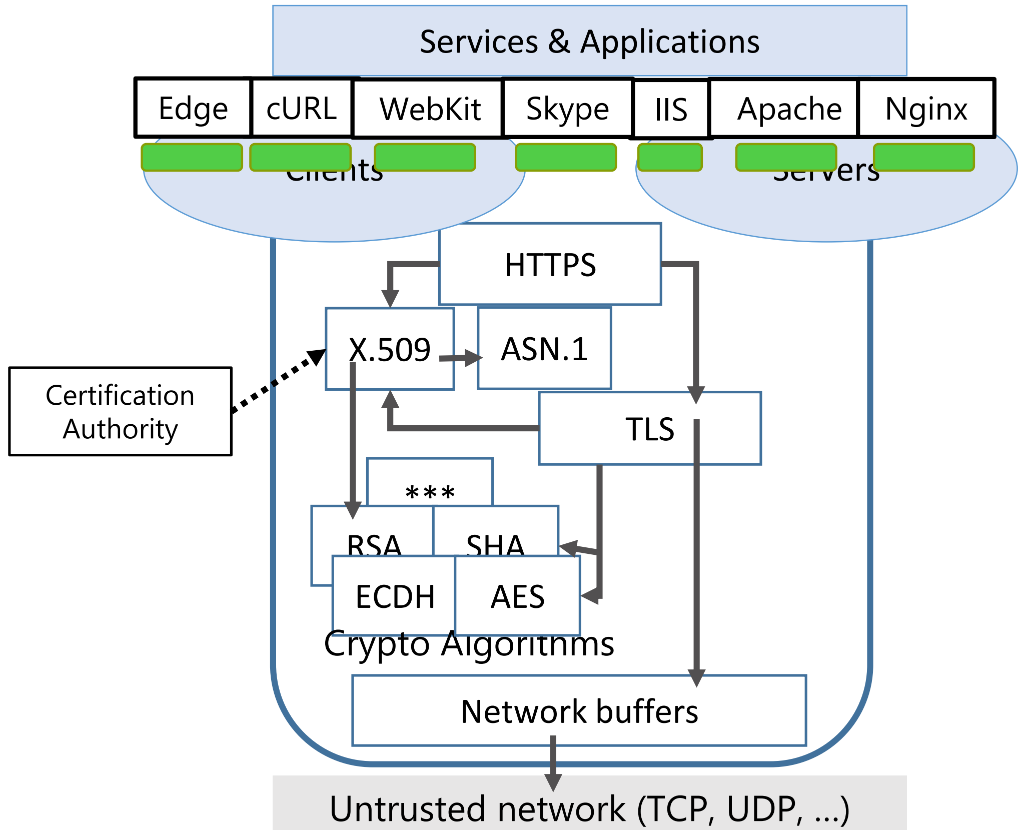
Side Channel Challenge (Attacks)

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

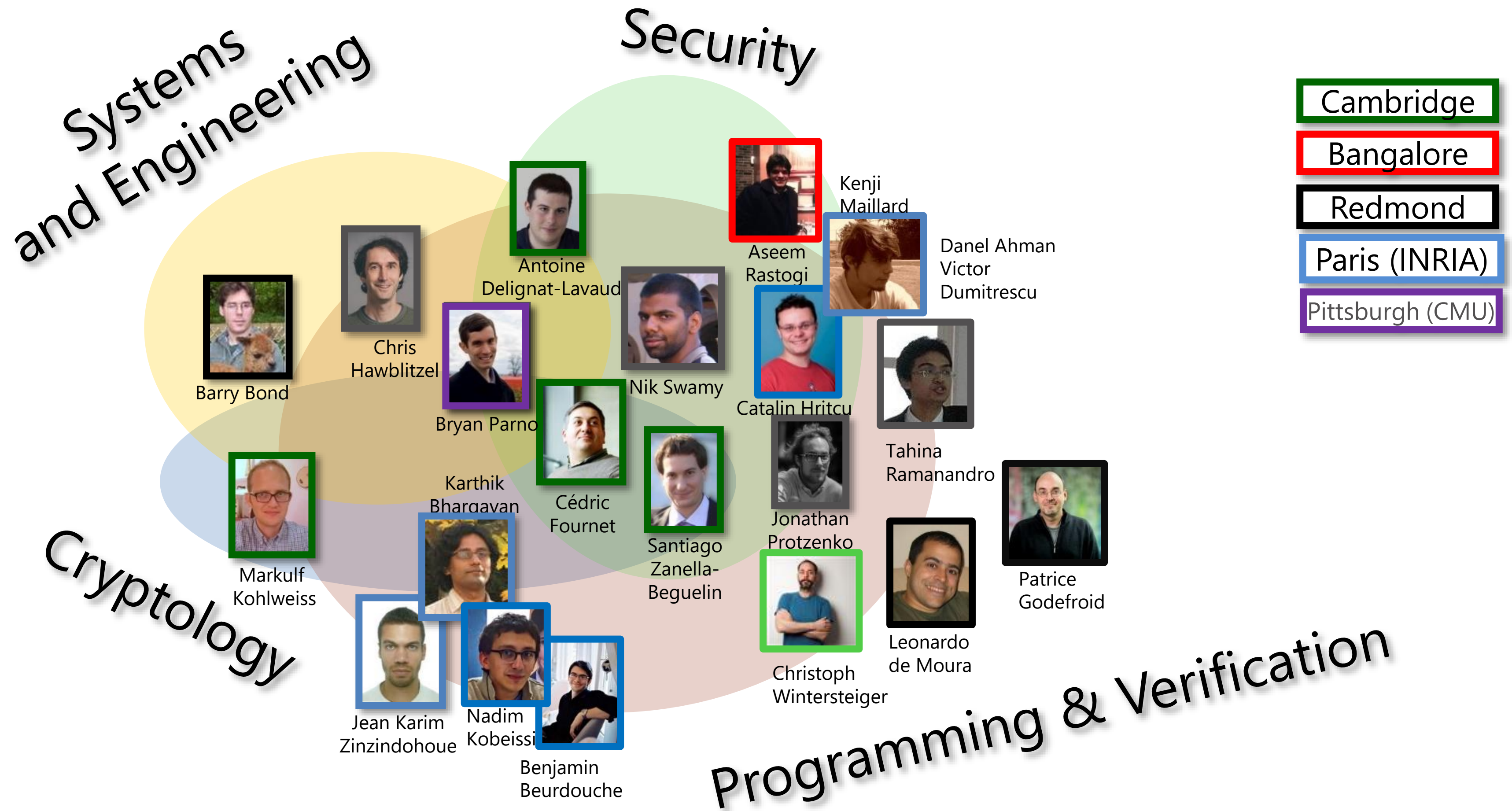


Verified Components for the HTTPS Ecosystem

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



Team Everest



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 ☹

Software safety & security (implementation)

Focus on performance, error handling, operational security

Security vulnerabilities & patches

RFCs (informal specs)

Focus on wire format, flexibility, and interoperability

Security is considered, not specified

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

kreMLin

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

C/C++

gcc, compcert, clang, msvc

interop with rest of TLS/HTTPS ecosystem

production code

Concrete Applications

blog.mozilla.org/security/2017/09/13/verified-cryptography-firefox-57

mozilla

Mozilla Security Blog

SEP 13 2017

Verified cryptography for Firefox 57

Benjamin Beurdouche

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.

Benjamin Beurdouche

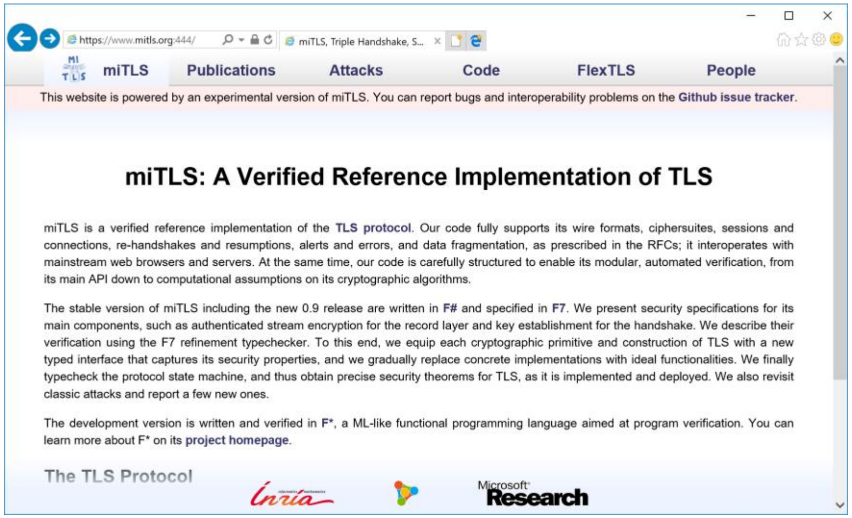
Mozillian INRIA Paris - Prosecco team

More from Benjamin Beurdouche »

Categories

- Announcements
- Automated Testing
- BrowserID

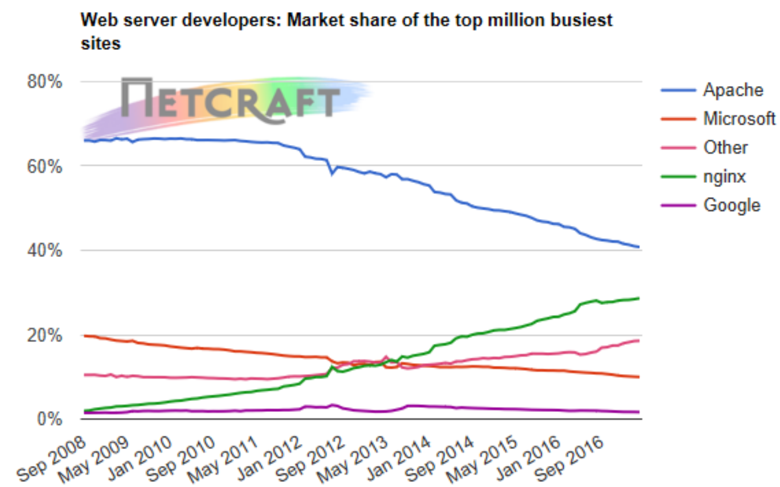
Client: IE



We integrate miTLS & its verified crypto with Internet Explorer.

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

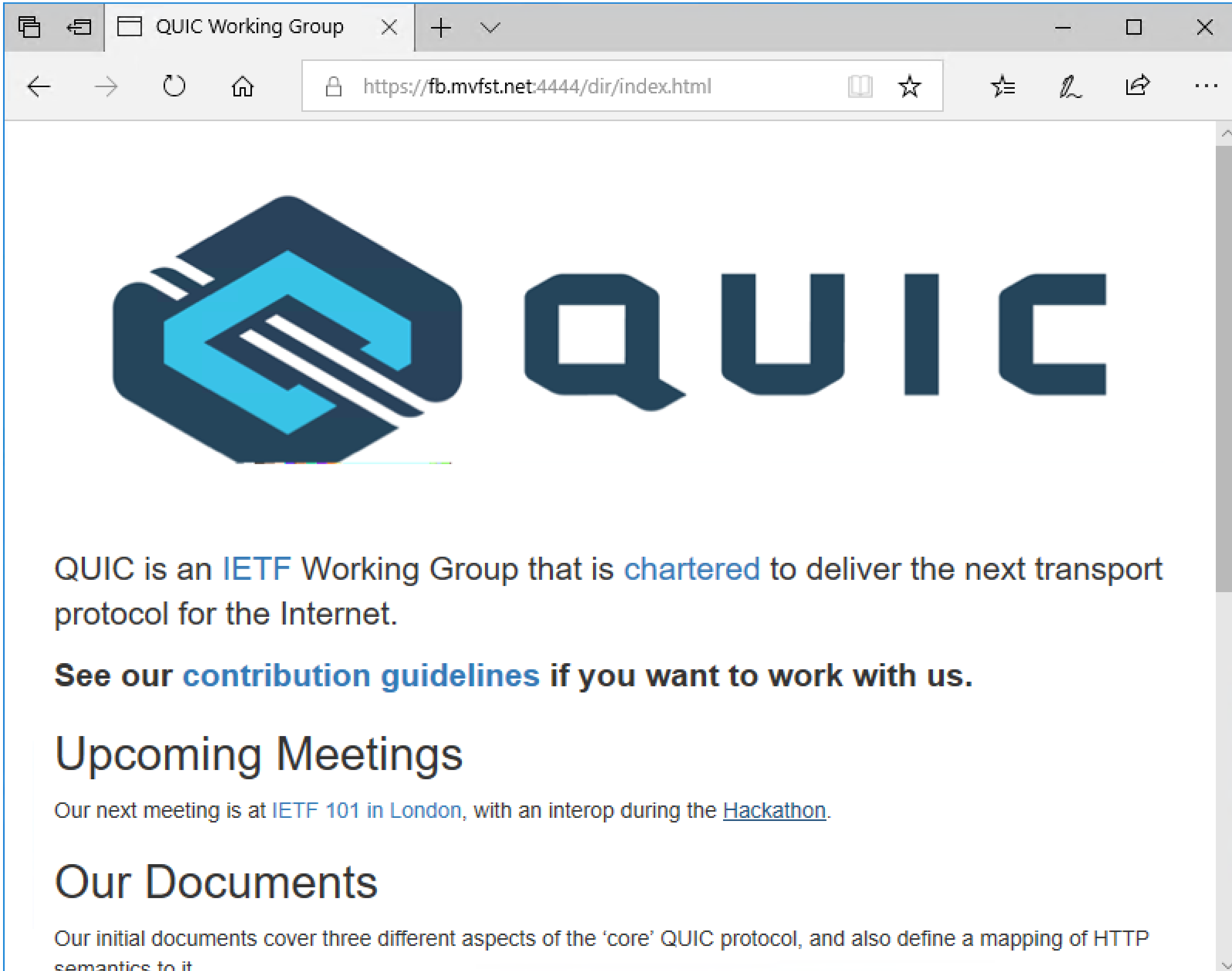
Server: nginx



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



Modelling concrete cryptographic security in F*

Cryptographic Integrity:

Message Authentication Codes (MAC)

```
module HMAC_SHA256 (* plain *)

type key
type msg = bytes
type tag = lbytes 32

val keygen: unit → St key
val mac: key → msg → tag
val verify: key → msg → tag → bool
```

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))
```

This **ideal interface** uses a log to specify security

Great for F* verification.

Unrealistic: tags can be guessed

Idealization

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

- Flip a coin $\boldsymbol{b}$
- Let $\boldsymbol{F}$ be the ideal function if head, the real one if tail
- Let $\boldsymbol{A}$ be a a program that can call $\boldsymbol{F}$ and tries to guess $\boldsymbol{b}$
- The advantage of $\boldsymbol{A}$ is $|\boldsymbol{Pr}(\boldsymbol{A}() = \boldsymbol{b}) - \mathbf{1/2}|$
- The functionality of $\boldsymbol{\epsilon}$-secure if $\mathrm{Adv}(\mathrm{A}) \leq \boldsymbol{\epsilon}$

Cryptographic Integrity: UF-CMA security (2/3)

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)
```

Our ideal interface reflects the security of a **chosen-message game** [Goldwasser'88]

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

Game $\mathsf{Mac1}^b(\mathsf{MAC})$

$k \xleftarrow{\$} \mathsf{MAC.keygen}(\varepsilon);\ \mathit{log} \leftarrow \bot$
return $\{\mathsf{Mac}, \mathsf{Verify}\}$

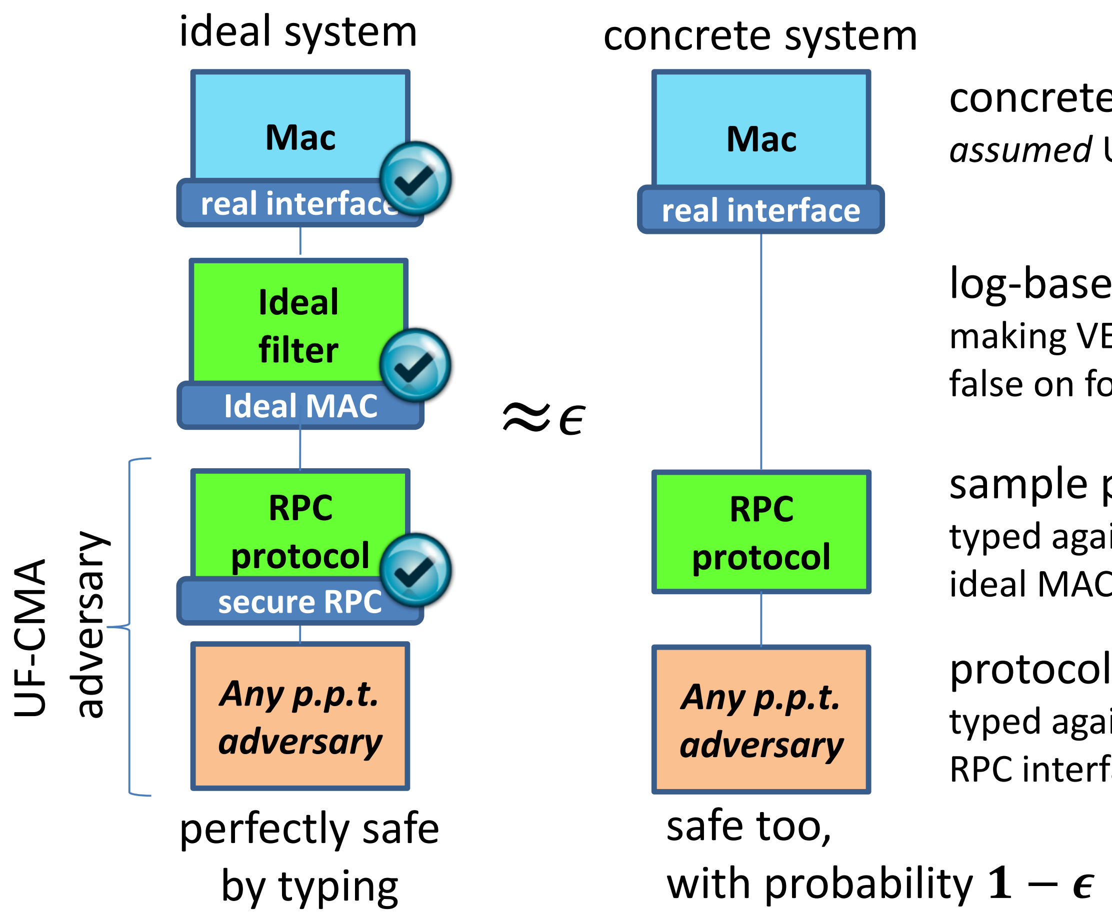
Oracle $\mathsf{Verify}(m^\star, t^\star)$

if b **return** $\mathit{log} = (m^\star, t^\star)$
return $\mathsf{MAC.verify}(k, m^\star, t^\star)$

Oracle $\mathsf{Mac}(m)$

if $\mathit{log} \neq \bot$ **return** $\bot$
$t \leftarrow \mathsf{MAC.mac}(k, m)$
if $b \wedge r$
  $t \xleftarrow{\$} \mathsf{byte}^{\mathsf{MAC}.\ell_t}$
$\mathit{log} \leftarrow (m, t)$
return t

Cryptographic Integrity: UF-CMA security (3/3)



concrete algorithm
assumed UF-CMA computationally

log-based error correction
making VERIFY returns
false on forgeries

sample protocol
typed against
ideal MAC interface

protocol adversary
typed against
RPC interface

Cryptographic Integrity: Two styles for ideal MACs

```
module MAC (* stateful *)

type key
val log: mem → key → Seq msg

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)

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

```
module MAC (* logical *)

type property: msg → Type
type key (p:prop)

val keygen:
  #p:property → St (key p)

val mac:
  #p: property → key p →
  m:msg {p m} → St tag

val verify:
  #p:property → key p → m:msg → tag →
  St (b:bool {b ⟹ p m}
```

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

type log (p:property) =
   mref (seq (m:msg {p})) grows
```

Authenticated Encryption

Cryptographic Confidentiality
Indistinguishability under Chosen-Plaintext Attacks

```
module Plain

abstract type plain = bytes

val repr: p:plain{¬ ideal} → Tot bytes
val coerce: r:bytes{¬ ideal} → Tot plain

let repr p = p
let coerce r = r

val length: plain → Tot ℕ
let length p = length p
```

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

AE.Ideal.fst

File Edit Options Buffers Tools F✪ Help

```
module AE.Game

let encrypt (k:key, p: plaintext) =
  if b then
    let c = randomBytes (length p) in
    k.log := k.log ++ (c ↝ p);
    c
  else
    AE.encrypt k.real p

let decrypt (k:key, c: ciphertext) =
  if b then
    Map.lookup !k.log c
  else
    AE.decrypt k.real c
```

-:**- AE.Ideal.fst Top L15 (F✪ +3 FlyC- company ElDoc)

We capture its security using types to keep track of the content of the log

Game $\mathsf{Ae}(\mathcal{A}, \mathsf{AE})$

$b \xleftarrow{\$} \{0,1\};\ L \leftarrow \varnothing;\ k \xleftarrow{\$} \mathsf{AE.keygen}()$

$b' \leftarrow \mathcal{A}^{\mathsf{Encrypt},\mathsf{Decrypt}}();\ \mathbf{return}\ (b \stackrel{?}{=} b')$

Oracle $\mathsf{Encrypt}(p)$

$\mathbf{if}\ b\ \ \mathbf{then}\ c \xleftarrow{\$} \mathsf{byte}^{\ell_c};\ L[c] \leftarrow p$

$\mathbf{else}\ \ c \leftarrow \mathsf{AE.encrypt}\ k\ p$

$\mathbf{return}\ c$

Oracle $\mathsf{Decrypt}(c)$

$\mathbf{if}\ b\ \ \mathbf{then}\ p \leftarrow L[c]$

$\mathbf{else}\ \ p \leftarrow \mathsf{AE.decrypt}\ k\ c$

$\mathbf{return}\ p$

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

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

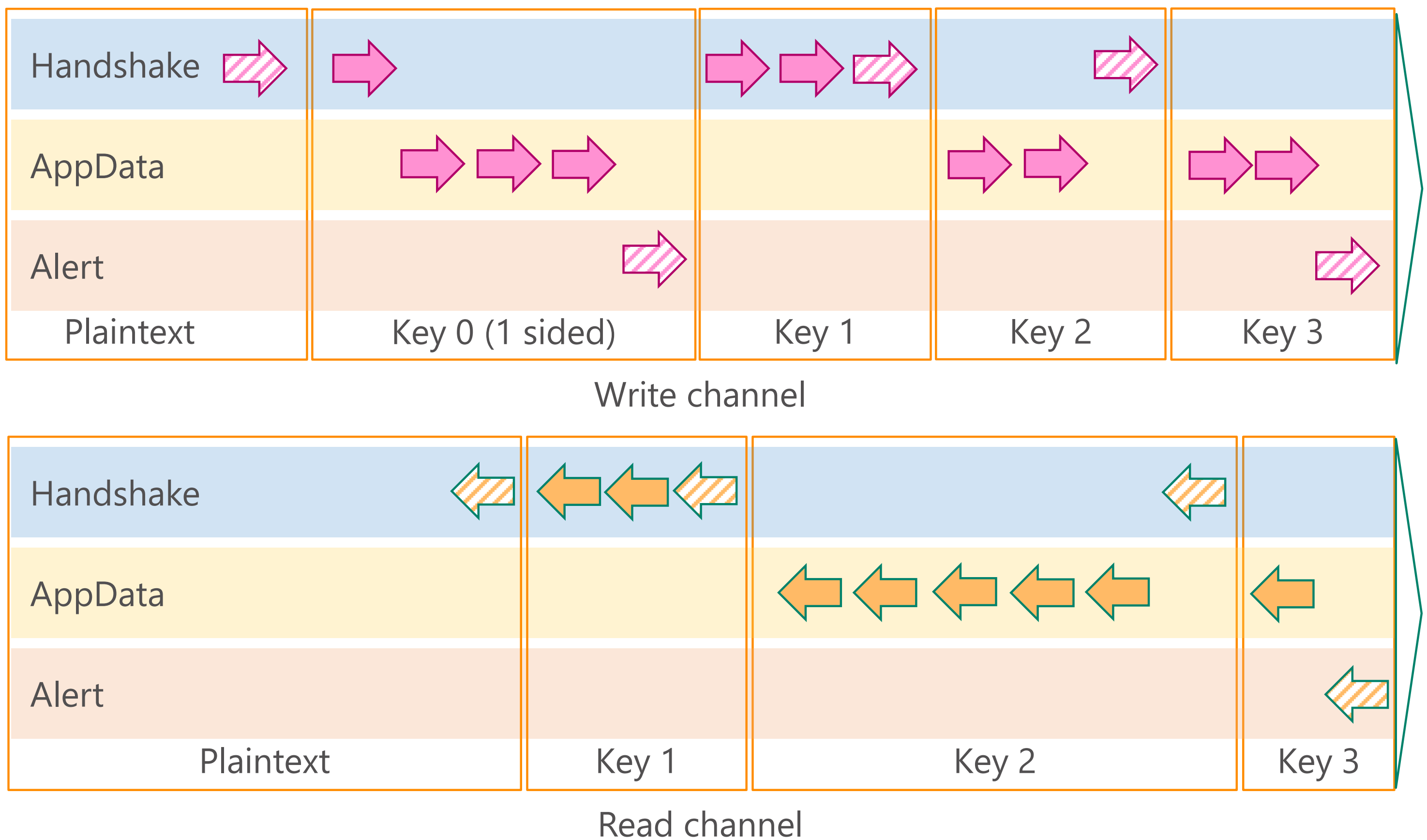
Standardized complications

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

Poor TLS track record

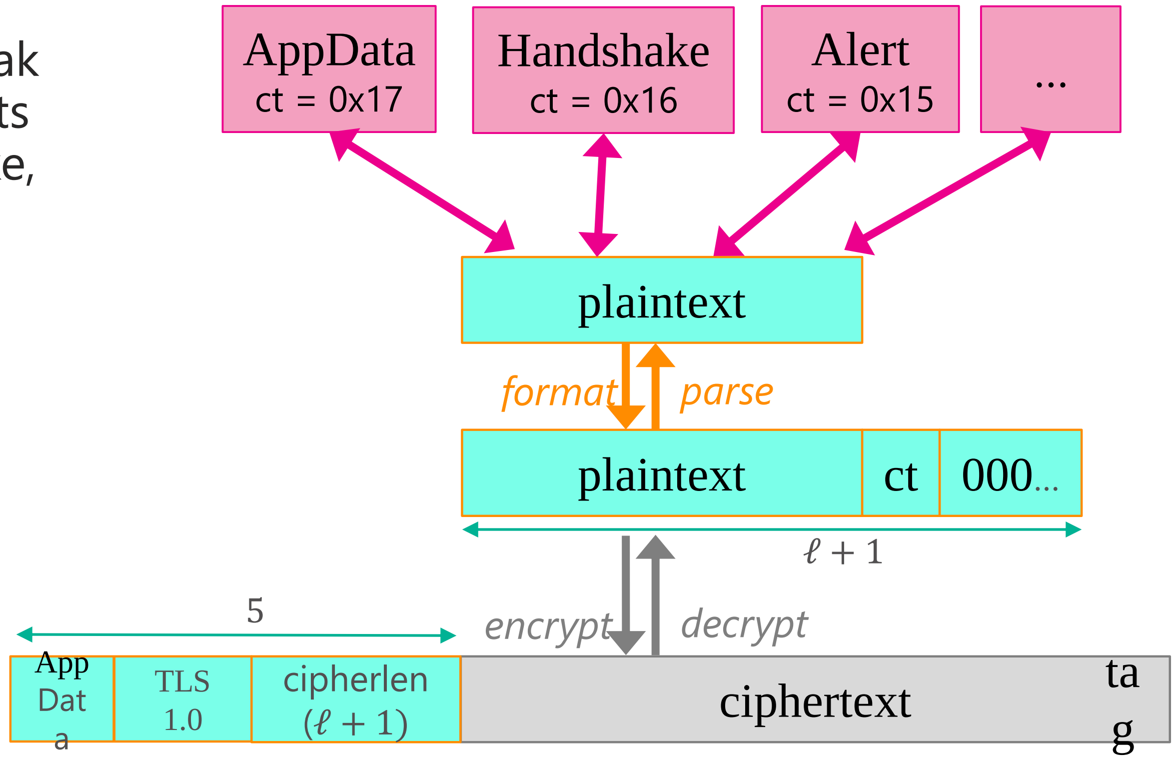
- Many implementation flaws
- Attacks on weak cryptography (MD5, SHA1, …)
- Attacks on weak constructions (MAC-Encode-then-Encrypt)
- Attacks on compression
- Persistent side channels
- Persistent truncation attacks

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

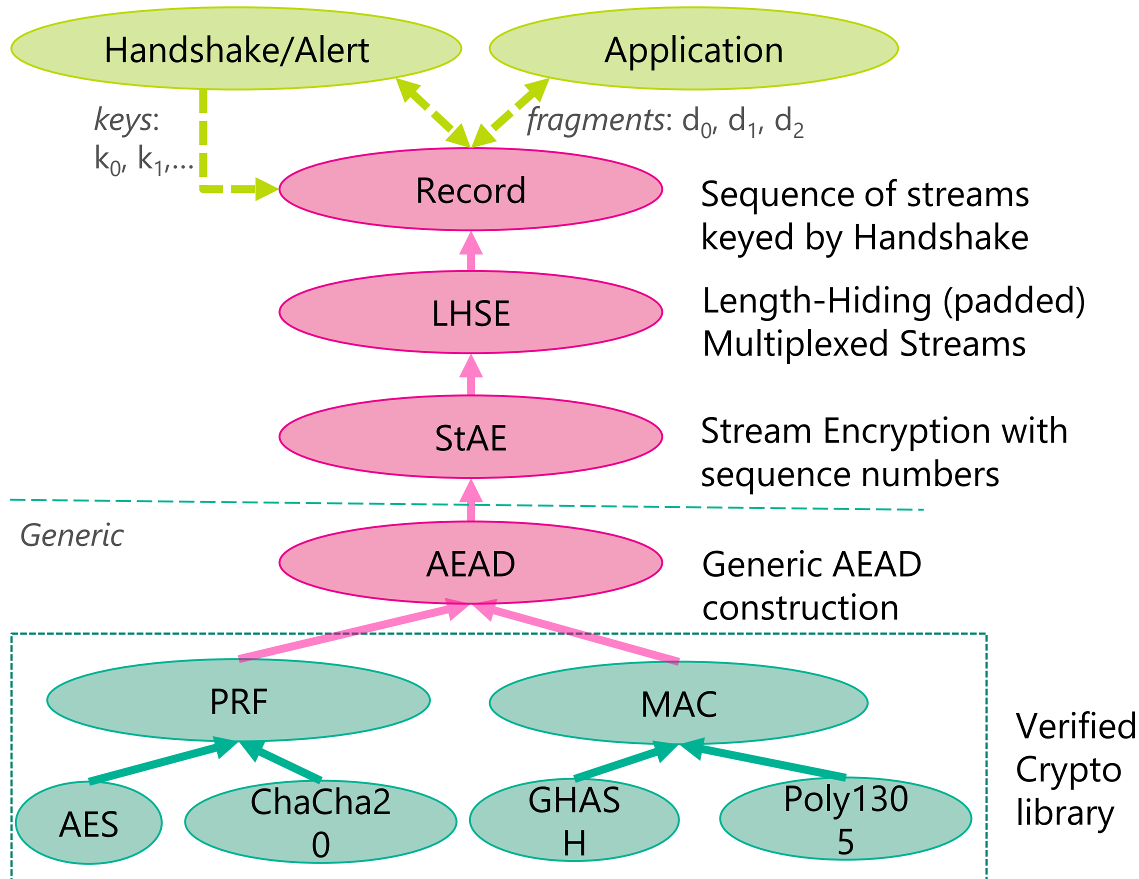


The TLS Record Layer (F*)

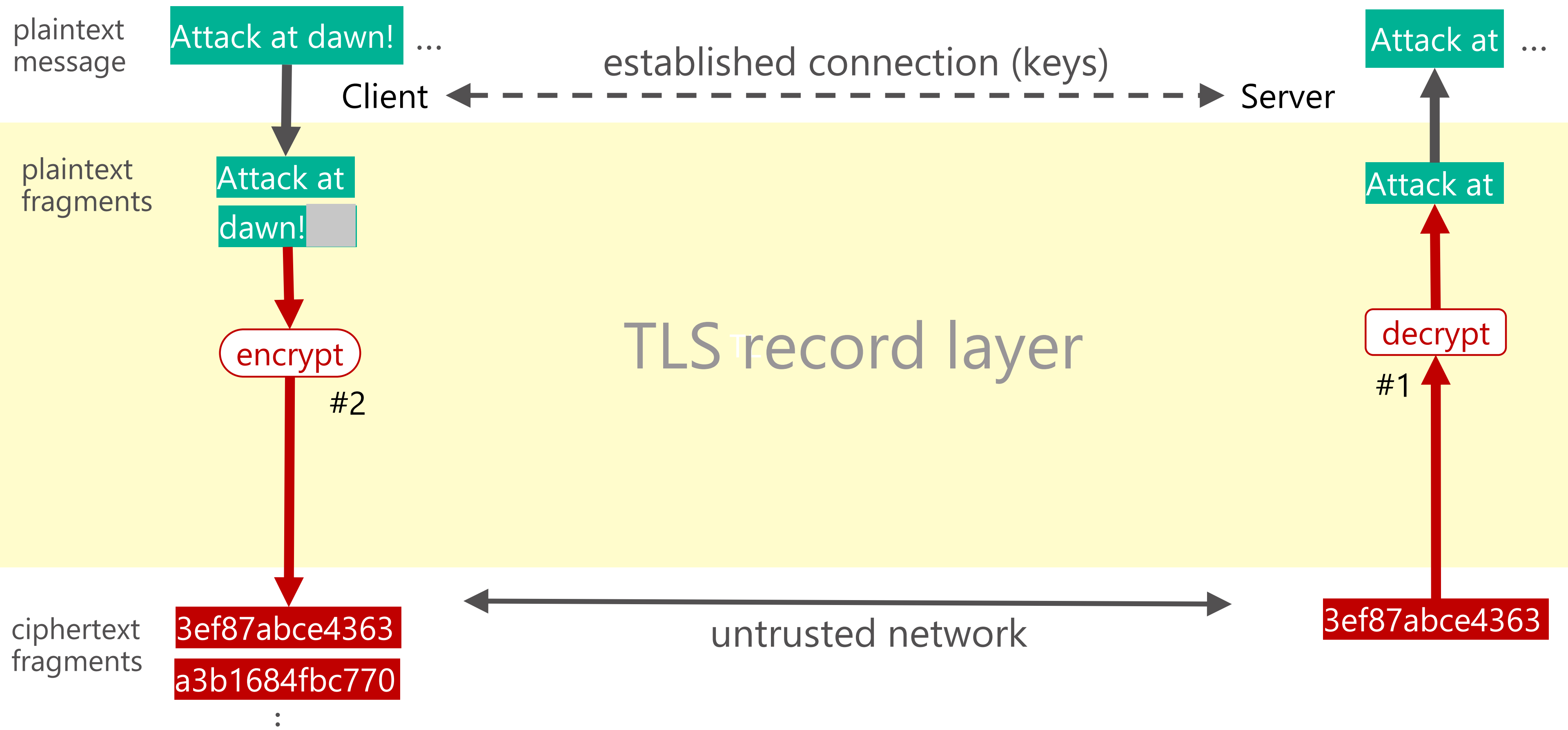
We model record-layer security using a game at every level of the construction.

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

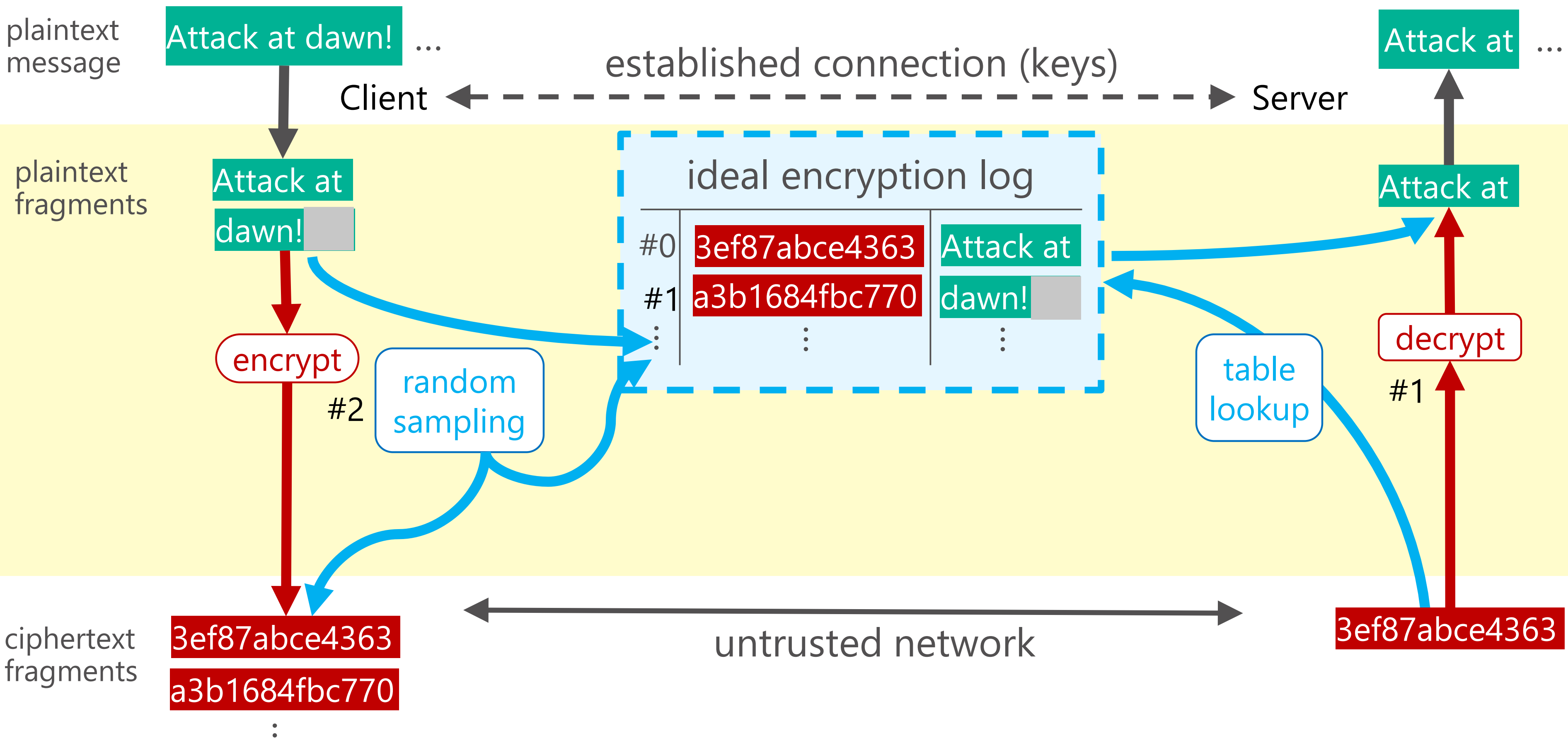
We obtain security guarantees at the top-level API for the TLS record layer

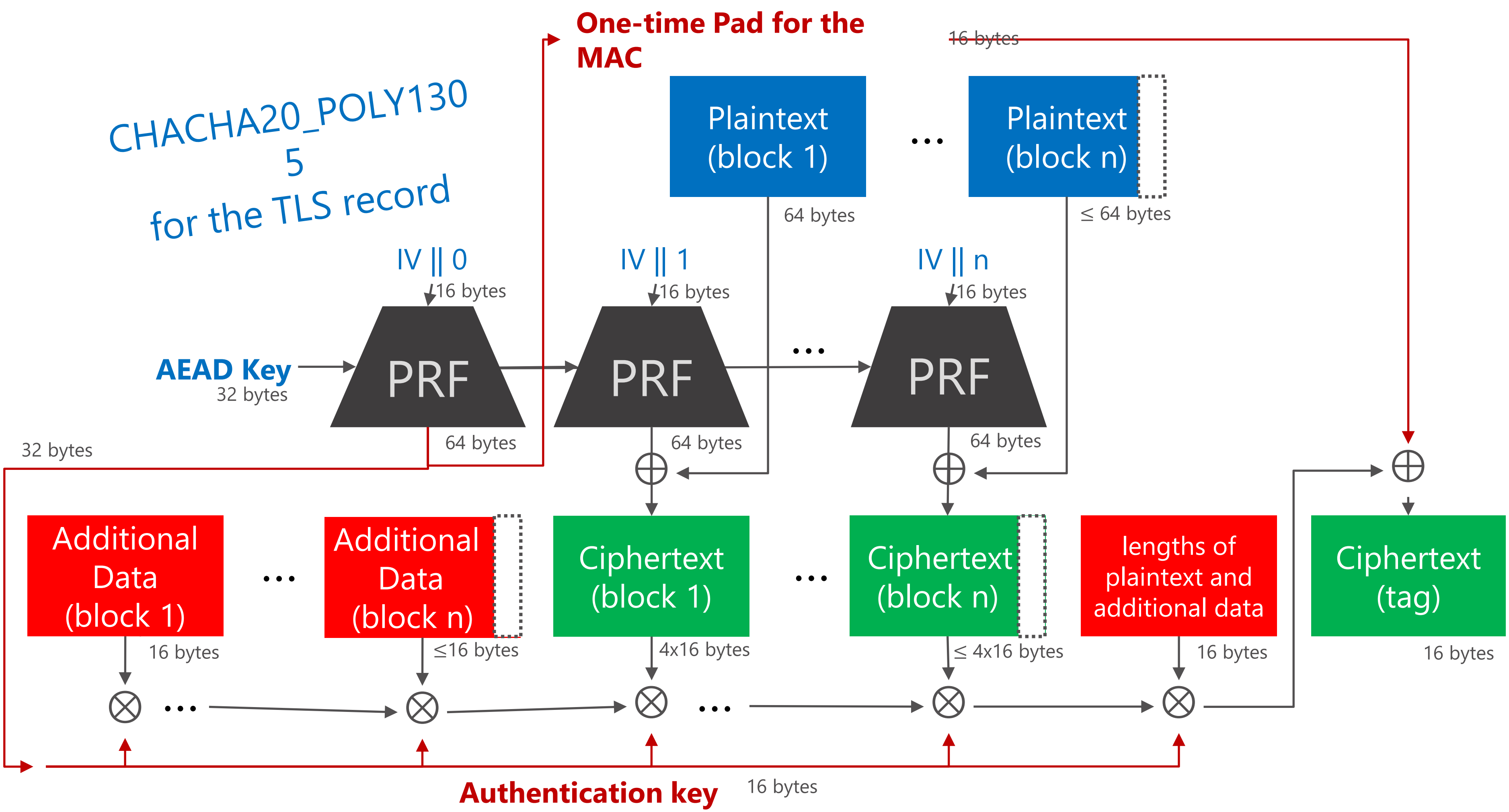


Stream Encryption: Security Definition



Stream Encryption: Security Definition





CHACHA20_POLY1305
for the TLS record
One-time Pad for the MAC
~~16 bytes~~
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
64 bytes
64 bytes
64 bytes
32 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)

$$\begin{array}{l}
\underline{\textbf{Game } \mathsf{UF\text{-}1CMA}(\mathcal{A}, \mathsf{MAC})} \\
k \xleftarrow{\$} \mathsf{MAC.keygen}(\varepsilon);\ \mathit{log} \leftarrow \bot \\
(m^\star, t^\star) \leftarrow \mathcal{A}^{\mathsf{Mac}} \\
\textbf{return } \mathsf{MAC.verify}(k, m^\star, t^\star) \\
\qquad \wedge\ \mathit{log} \neq (m^\star, t^\star)
\end{array}$$

$$\begin{array}{l}
\underline{\textbf{Oracle } \mathsf{Mac}(m)} \\
\textbf{if } \mathit{log} \neq \bot \textbf{ return } \bot \\
t \leftarrow \mathsf{MAC.mac}(k, m) \\
\mathit{log} \leftarrow (m, t) \\
\textbf{return } t
\end{array}$$

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

Ciphers (IND-PRF)

$$\begin{array}{l}
\underline{\textbf{Game } \mathsf{Prf}^b(\mathsf{PRF})} \\
T \leftarrow \varnothing \\
k \xleftarrow{\$} \mathsf{PRF.keygen}() \\
\textbf{return } \{\mathsf{Eval}\}
\end{array}$$

$$\begin{array}{l}
\underline{\textbf{Oracle } \mathsf{Eval}(m)} \\
\textbf{if } T[m] = \bot \\
\quad \textbf{if } b \textbf{ then } T[m] \xleftarrow{\$} \mathsf{byte}^{\ell_b} \\
\quad \textbf{else } T[m] \leftarrow \mathsf{PRF.eval}(k, m) \\
\textbf{return } T[m]
\end{array}$$

Assumed for AES and Chacha20

Stream Encryption: Assumptions

One-Time MACs (INT-CMA1)

$$\begin{array}{l}
\underline{\textbf{Game } \mathsf{UF\text{-}1CMA}(\mathcal{A}, \mathsf{MAC})} \\
k \xleftarrow{\$} \mathsf{MAC.keygen}(\varepsilon);\ \ \mathit{log} \leftarrow \bot \\
(m^\star, t^\star) \leftarrow \mathcal{A}^{\mathsf{Mac}} \\
\textbf{return } \mathsf{MAC.verify}(k, m^\star, t^\star) \\
\qquad \wedge\ \mathit{log} \neq (m^\star, t^\star)
\end{array}
\qquad
\begin{array}{l}
\underline{\textbf{Oracle } \mathsf{Mac}(m)} \\
\textbf{if } \mathit{log} \neq \bot \textbf{ return } \bot \\
t \leftarrow \mathsf{MAC.mac}(k, m) \\
\mathit{log} \leftarrow (m, t) \\
\textbf{return } t
\end{array}$$

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)

$$\begin{array}{l}
\underline{\textbf{Game } \mathsf{Prf}^b(\mathsf{PRF})} \\
T \leftarrow \varnothing \\
k \xleftarrow{\$} \mathsf{PRF.keygen}() \\
\textbf{return } \{\mathsf{Eval}\}
\end{array}
\qquad
\begin{array}{l}
\underline{\textbf{Oracle } \mathsf{Eval}(m)} \\
\textbf{if } T[m] = \bot \\
\quad \textbf{if } b \textbf{ then } T[m] \xleftarrow{\$} \mathsf{byte}^{\ell_b} \\
\quad \textbf{else } T[m] \leftarrow \mathsf{PRF.eval}(k, m) \\
\textbf{return } T[m]
\end{array}$$

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!

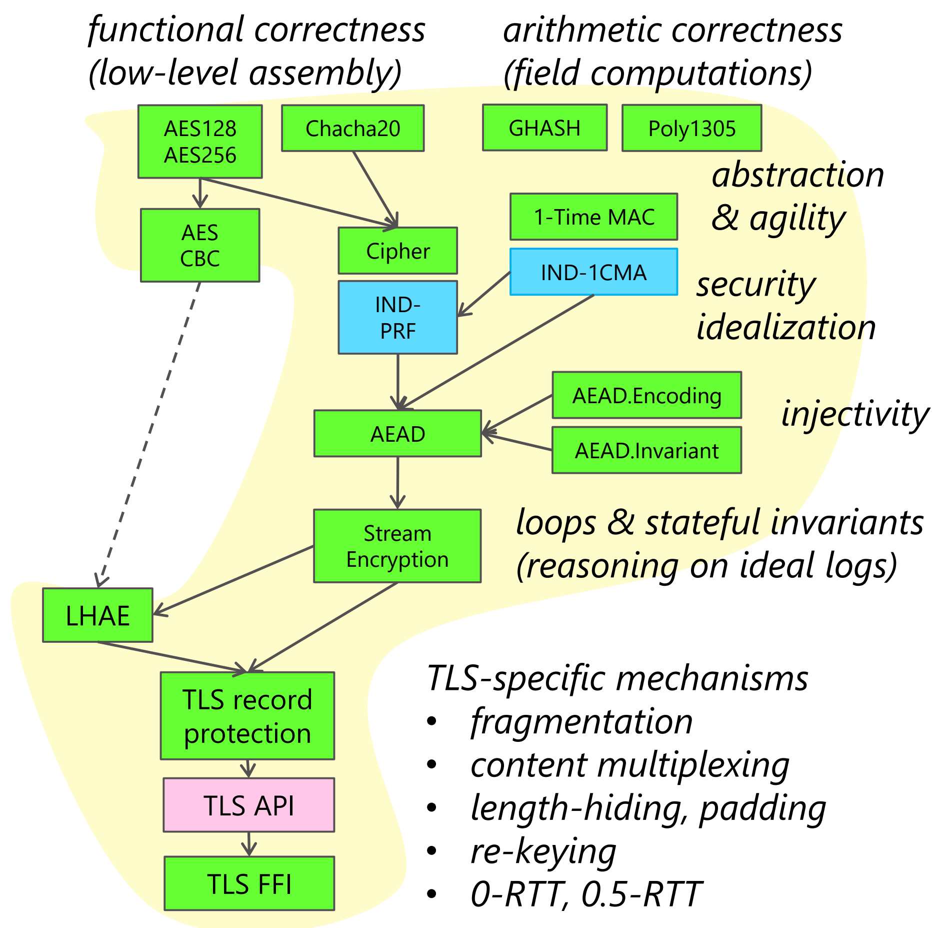
Given

- 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	$\epsilon_{\mathsf{Lhse}}(\mathcal{A}[q_e, q_d]) \leq$
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 $q_d \leq 2^{60}$ per stream)

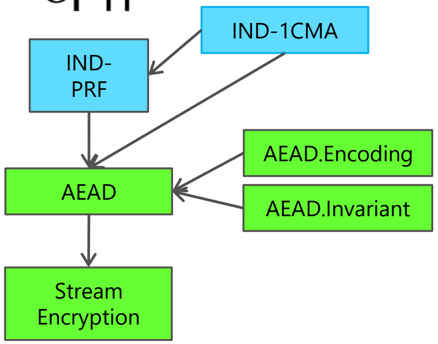
q_e is the number of encrypted records;
q_d is the number of chosen-ciphertext decryptions;
q_b is the total number of blocks for the PRF

Standard crypto assumption

ϵ_{Prf}

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

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



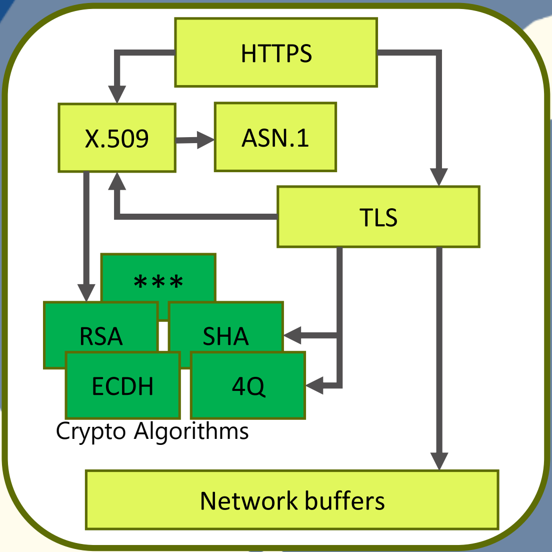
$$\epsilon_{\mathsf{Lhse}}(\mathcal{A}[q_e, q_d]) = \epsilon_{\mathsf{Prf}} + \epsilon_{\mathsf{MMac1}}$$

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

Stream Encryption: 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

Verified High-Assurance Crypto Libraries



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."

```
These produce wrong results. The first example d
the other three also on 64 bit.
```

.pl

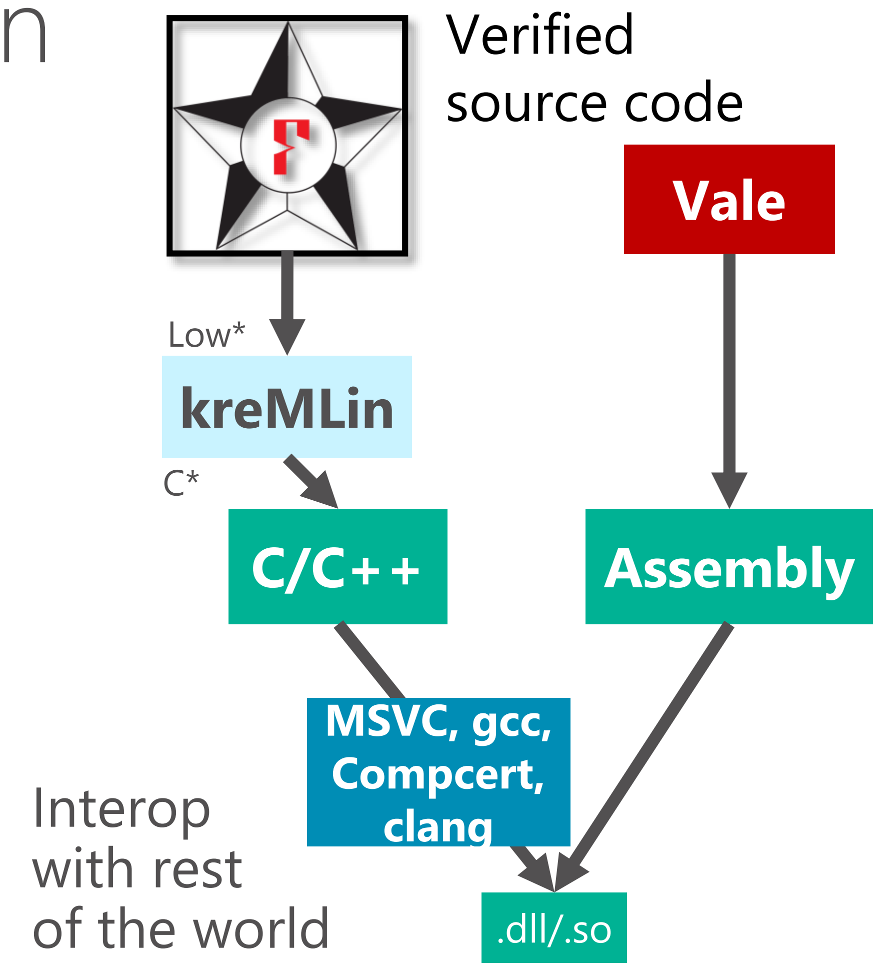
```
You know the drill. See the attached poly1305_test2.c.

$ OPENSSL_ia32cap=0 ./poly1305_test2
PASS
$ ./poly1305_test2
Poly1305 test failed.
got:      2637408fe03086ea73f971e3425e2820
expected: 2637408fe13086ea73f971e3425e2820

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
```

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

Crypto verification & compilation Toolchain

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 __ARMEB__
  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

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

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

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

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 $\approx \frac{|\vec{w}|}{2^{130}}$

Specifying, programming & verifying poly1305

Sample F* code: the **spec** for the multiplicative MAC used in TLS 1.3

Its verified optimized implementation for x64 takes 3K+ LOCs

Spec.Poly1305.fst

File Edit Options Buffers Tools Help

```
module Spec.Poly1305

(* Mathematical specification of multiplicative
   hash in the prime field 2^130 - 5  *)

let prime = 2^130 - 5

type elem = e:ℕ{e < prime}

let a +@ b = (a + b) % prime
let a  *@ b = (a × b) % prime

let encode (word:bytes {length w ≤ 8}): elem =
  2^(8 × length word) +@ little_endian word

let rec poly (text: seq bytes) (r: elem): elem =
  if Seq.length text = 0 then 0
  else encode (Seq.head text)) +@ poly (Seq.tail text) r) *@ r
```

-**- Spec.Poly1305.fst   Top L19   Git-dev (F✪ +3)

Sample crypto algorithm: poly1305

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

A typical 64-bit arithmetic implementation:

1. Represent elements of the prime field for $\mathrm{p} = 2^{130} - 5$ using **3 limbs** holding 42 + 44 + 44 bits in 64-bit registers
2. Use $(a.2^{130} + b)\ \%\ p = (a + 4a + b)\ \%\ p$ 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.

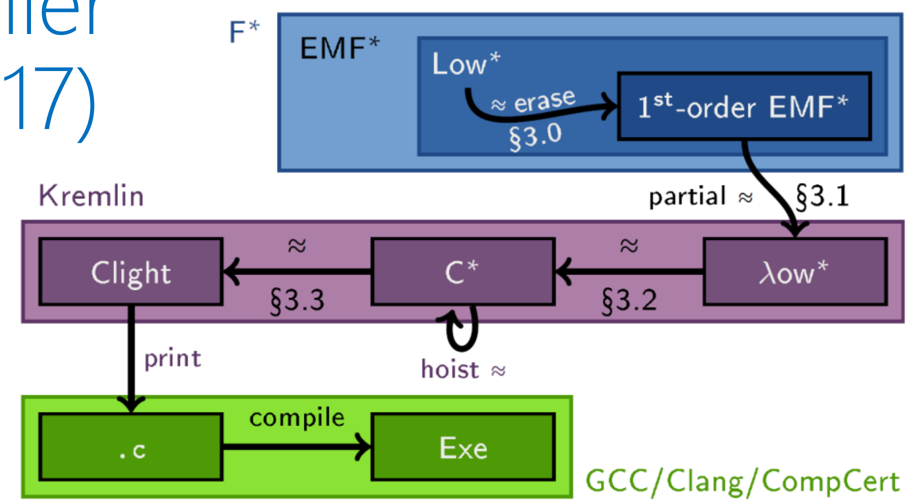
Low*: a subset of F* for safe C-style programming

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

Hacl.Impl.Poly1305_64.fst

File Edit Options Buffers Tools F★ Help

```
[@"substitute"]
val poly1305_last_pass_:
  acc:felem →
  Stack unit
    (requires (λ h → live h acc ∧ bounds (as_seq h acc) p44 p44 p42))
    (ensures (λ h0 _ h1 → live h0 acc ∧ bounds (as_seq h0 acc) p44 p44 p42
      ∧ live h1 acc ∧ bounds (as_seq h1 acc) p44 p44 p42
      ∧ modifies_1 acc h0 h1
      ∧ as_seq h1 acc == Hacl.Spec.Poly1305_64.poly1305_last_pass_spec_ (as_seq h0 acc)))

[@"substitute"]
let poly1305_last_pass_ acc =
  let a0 = acc.(0ul) in
  let a1 = acc.(1ul) in
  let a2 = acc.(2ul) in
  let open Hacl.Bignum.Limb 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_2 (v Hacl.Spec.Poly1305_64.p44m5);
  UInt.logand_lemma_2 (v Hacl.Spec.Poly1305_64.p44m1); UInt.logand_lemma_2 (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'
```

-:**- **Hacl.Impl.Poly1305_64.fst** 55% L394 Git-master (F★ FlyC- company ElDoc Wrap)

Poly1305_64.c

File Edit Options Buffers Tools C Help

```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 )0xfffffffffff;
  uint64_t r0 = a0 >> (uint32_t )44;
  uint64_t a1_ = (a10 + r0) & (uint64_t )0xfffffffffff;
  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 i0 = acc[0];
  uint64_t i1 = acc[1];
  uint64_t i0_ = i0 & (((uint64_t )1 << (uint32_t )44) - (uint64_t )1);
  uint64_t i1_ = 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 mask0 = FStar_UInt64_gte_mask(a00, (uint64_t )0xffffffffffb);
  uint64_t mask1 = FStar_UInt64_eq_mask(a1, (uint64_t )0xfffffffffff);
  uint64_t mask2 = FStar_UInt64_eq_mask(a2, (uint64_t )0x3ffffffffff);
  uint64_t mask = mask0 & mask1 & mask2;
  uint64_t a0_0 = a00 - ((uint64_t )0xffffffffffb & mask);
  uint64_t a1_0 = a1 - ((uint64_t )0xfffffffffff & mask);
  uint64_t a2_0 = a2 - ((uint64_t )0x3ffffffffff & mask);
  acc[0] = a0_0;
  acc[1] = a1_0;
  acc[2] = a2_0;
}
```

-:**- **Poly1305_64.c** 49% L272 Git-master (C/l company A

Performance for verified C code compiled from F*

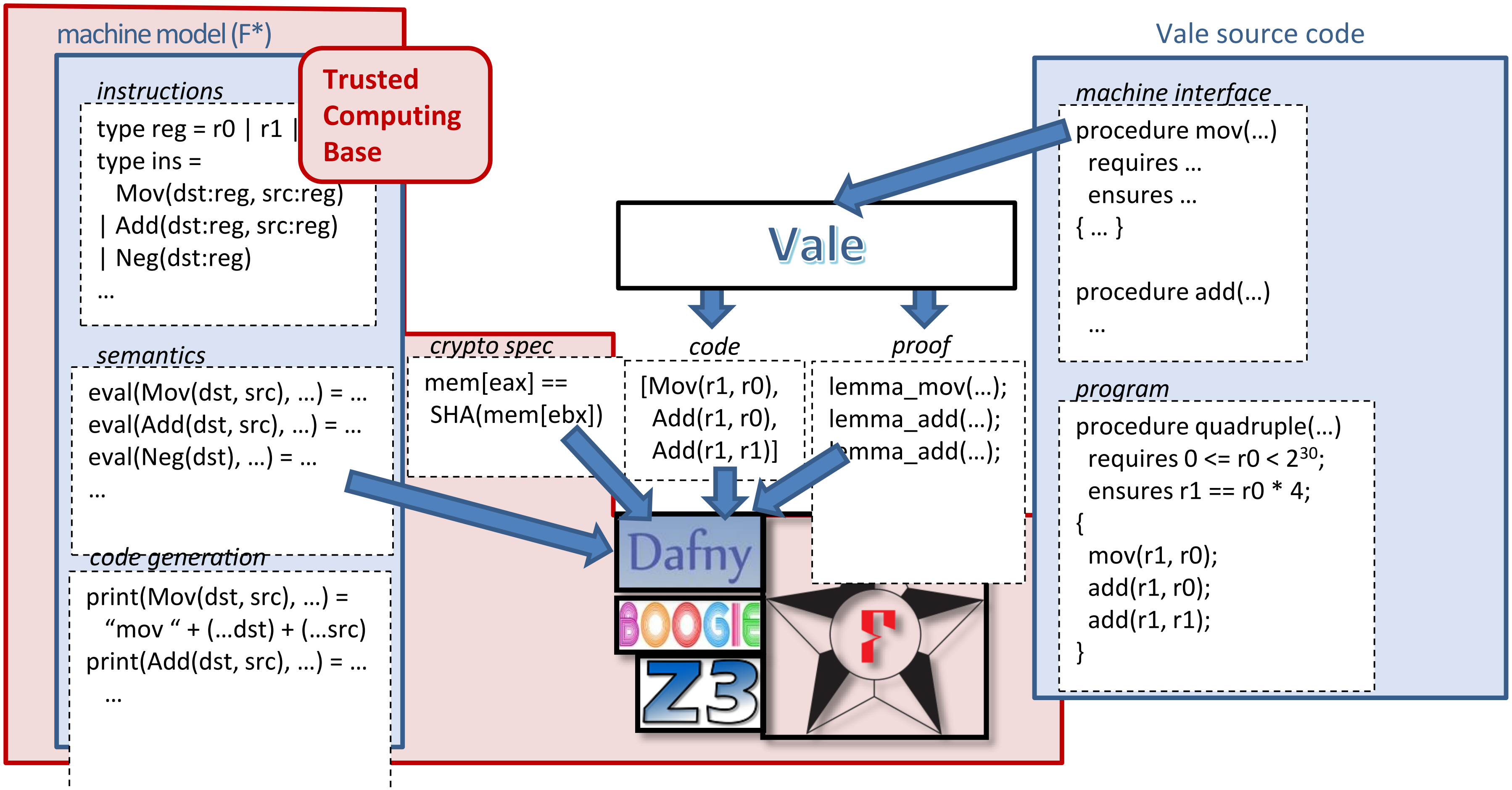
As fast as best hand-written
portable C implementations

Algorithm	HACL*	OpenSSL
ChaCha20	6.17 cy/B	8.04 cy/B
Poly1305	2.07 cy/B	2.16 cy/B
Curve25519	157k cy/mul	359k cy/mul

Still slower than best hand-written
assembly language implementations

Vale: extensible, automated assembly language verification (Usenix'17)

functional correctness & side-channel protection



OpenSSL Poly1305

raw.githubusercontent.c

raw.githubusercontent.com/openssl/openssl/mast

```
and     $d3,%rax
mov     $d3,$h2
shr     \$2,$d3
and     \$3,$h2
add     $d3,%rax
add     %rax,$h0
adc     \$0,$h1
adc     \$0,$h2
```

Bug! This carry was originally missing!

```
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);
    ...
}
```

Vale Poly1305

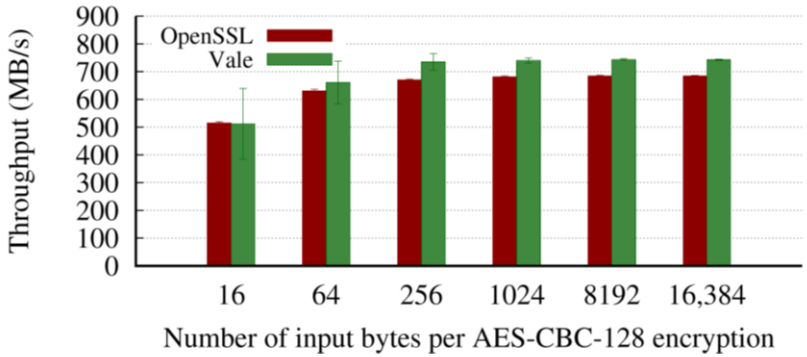
Vale Poly1305

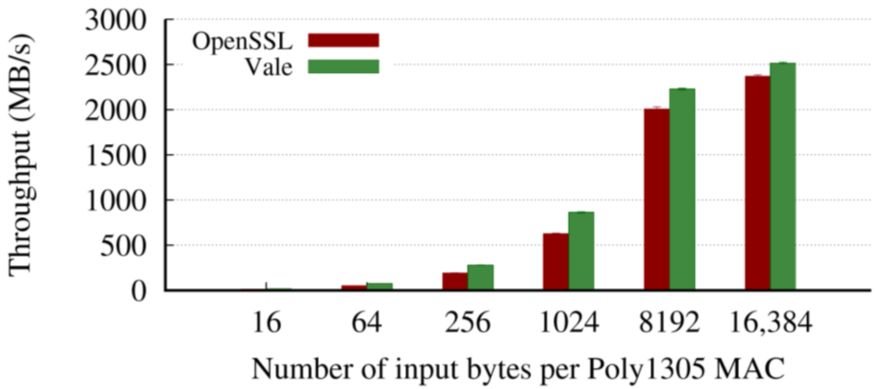
```
procedure poly1305_reduce() returns(ghost hOut:int)
    let
        n := 0x1_0000_0000_0000_0000;
        p := 4 * n * n - 5;
        hIn := (n * n) * d3 + n * h1 + h0;
        d3 @= r10; h0 @= r14; h1 @= rbx; h2 @= rbp;
    modifies
        rax; r10; r14; rbx; rbp; efl;
    requires
        d3 / 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)…Adc64Wrap(h2, 0);
    ghost var h10 := n * old(h1) + old(h0);
    hOut := h10 + rax + (old(d3) % 4) * (n * n);
    lemma_poly_reduce(n, p, hIn, old(d3), h10, rax, hOut); }
```

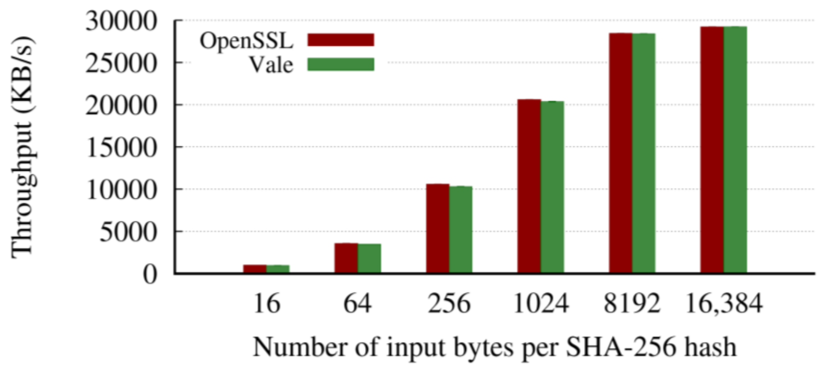
```
And64(rax, d3);
Mov64(h2, d3);
Shr64(d3, 2);
And64(h2, 3);
Add64Wrap(rax, d3);
Add64Wrap(h0, rax);
Adc64Wrap(h1, 0);
Adc64Wrap(h2, 0);
```

Performance: OpenSSL vs. Vale

- AES: OpenSSL with SIMD, AES-NI
- Poly1305 and SHA-256: OpenSSL non-SIMD assembly language (same assembly for OpenSSL, Vale)







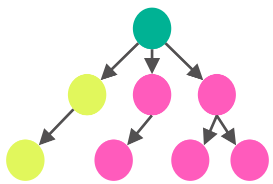
Verified interoperability between Low* and Vale

(Sneak peek of work in progress)

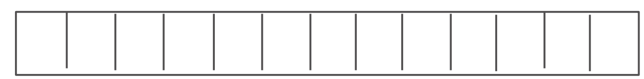
Goals: **End-to-end** functional correctness and side-channel resistance

Reconcile the memory models of Low* and Vale

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



(Each of these regions maps references to values)



Low* has a structured memory model

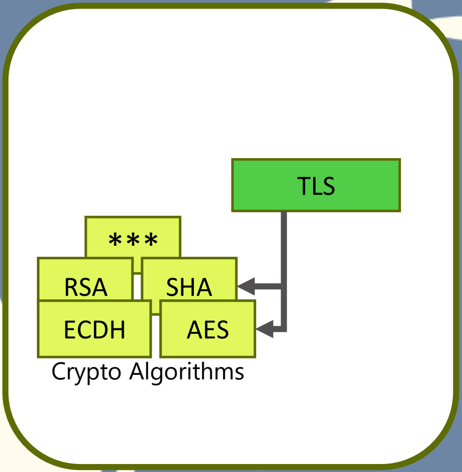
Vale memory is a flat array of bytes

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

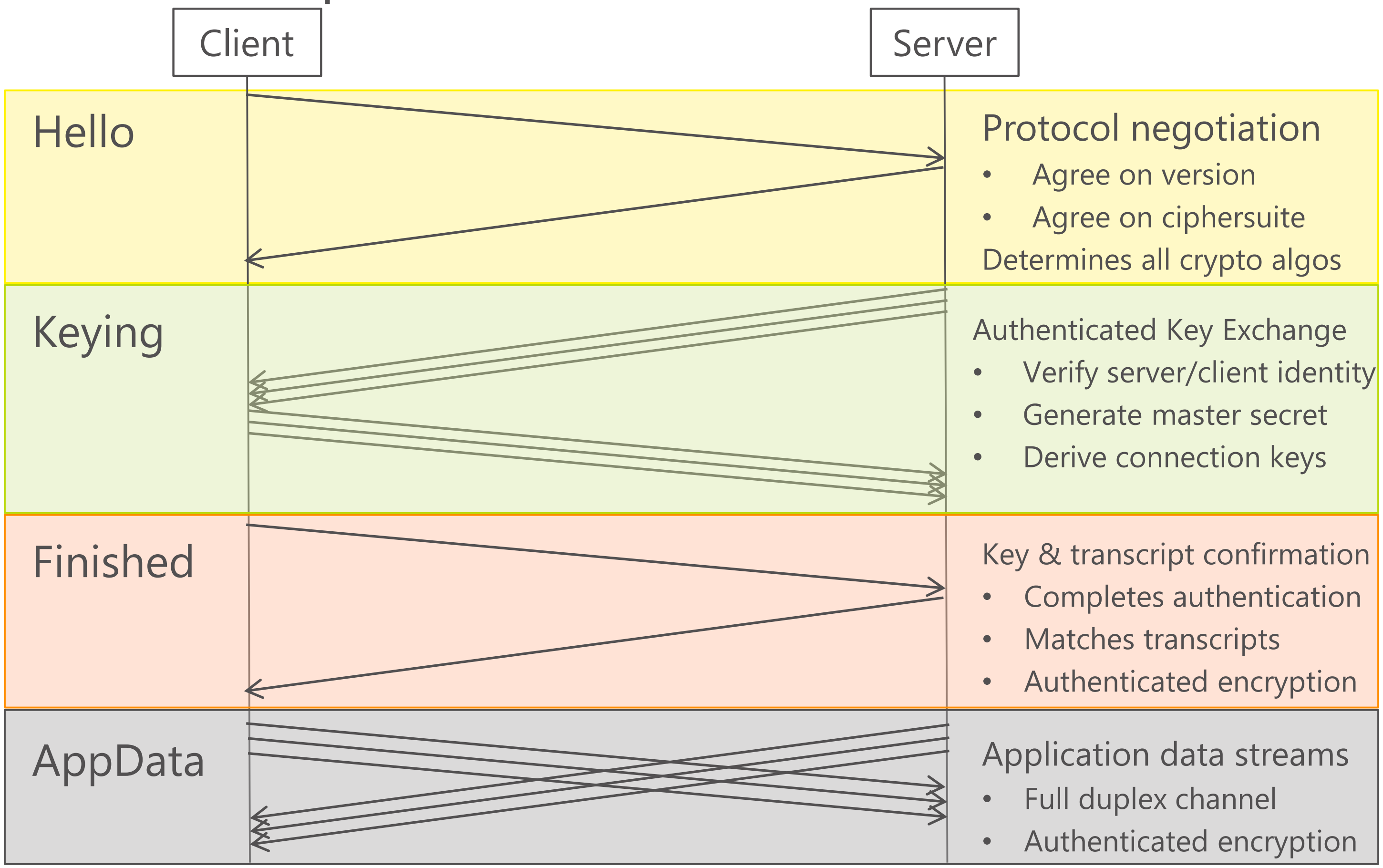
Compose the two specs

Reflect changes performed by the Vale code in the structured view, *allowing for temporary inconsistencies*

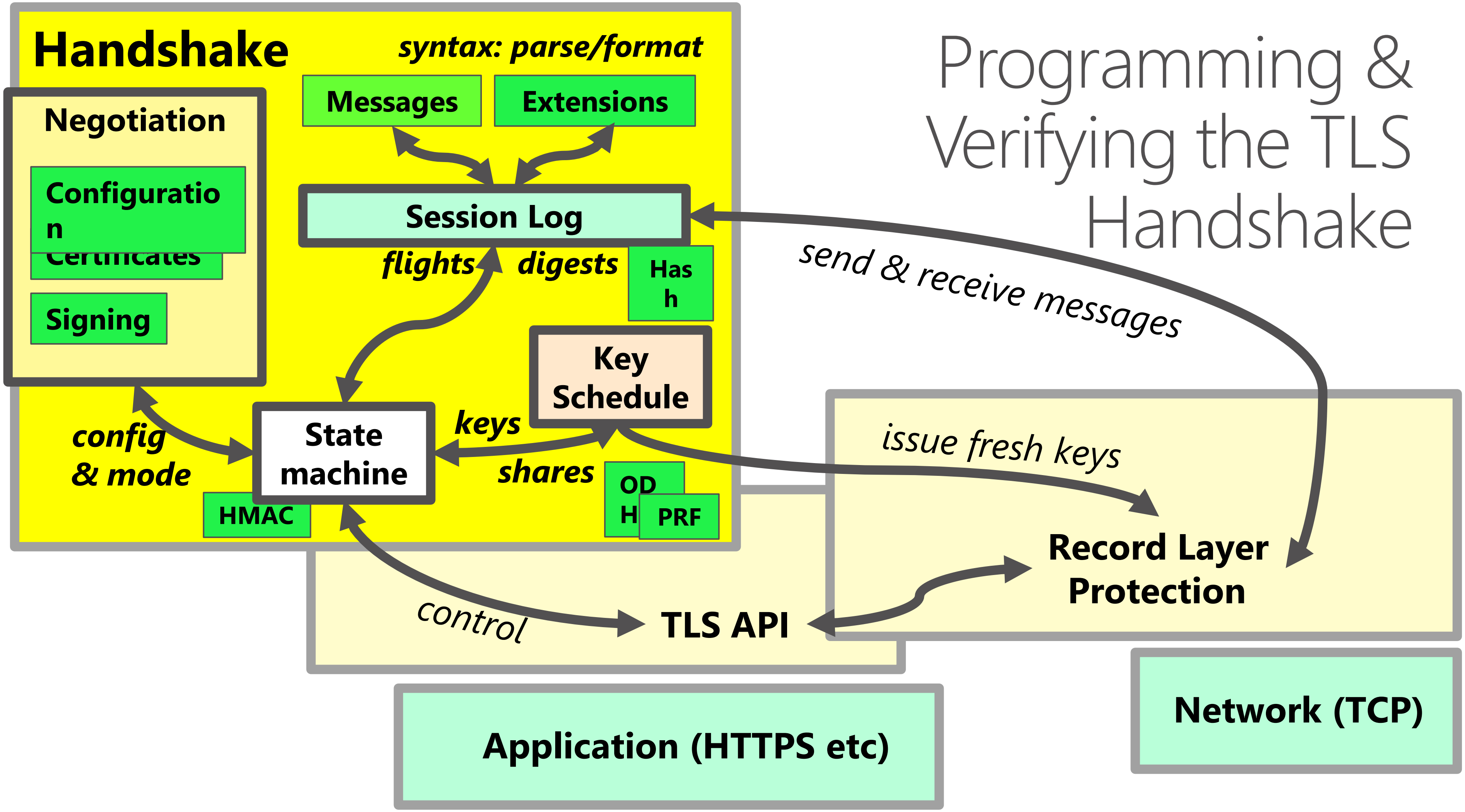
TLS 1.3 Handshake (Outline)



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
```

inverse properties

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

high-level type

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

high-level formatter

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

```
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

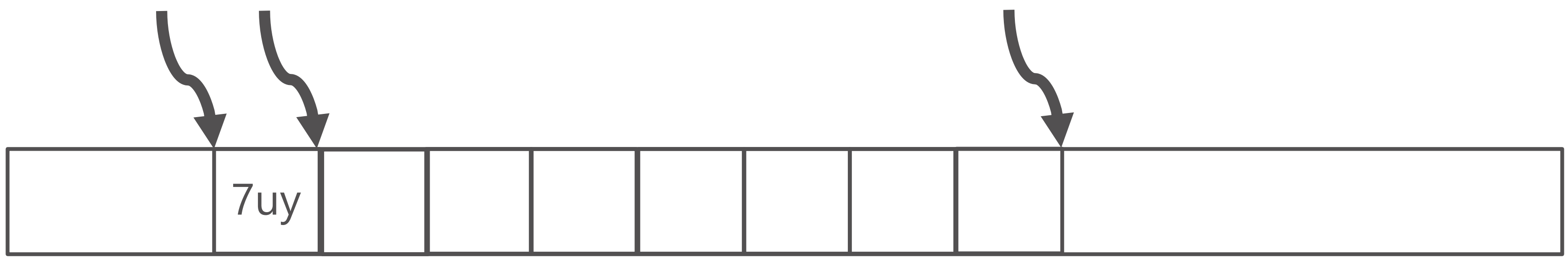
low-level validator

```
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)
```

Negotiation (highlights)

Flexibility vs security

Many standardized TLS versions, algorithms, constructions, extensions.

Even the draft number is negotiated!

New design (draft#17)

Backward compatibility

Critical for adoption and deployment

TLS 1.3 must support prior version negotiation, must "look like" TLS 1.2 for peers that do not understand TLS 1.3

Delicate coding & testing

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*
ClientKeyExchange
CertificateVerify*
[ChangeCipherSpec]
Finished                 -------->
                                  [ChangeCipherSpec]
                         <--------          Finished
Application Data         <------->  Application Data
```

TLS 1.2 (Abbreviated Handshake)

```
ClientHello              -------->
                                         ServerHello
                                  [ChangeCipherSpec]
                         <--------          Finished
[ChangeCipherSpec]
Finished                 -------->
Application Data         <------->  Application Data
```

TLS 1.3 (Full Handshake)

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

(EndOfEarlyData)
{Finished}               -------->
[Application Data]       <------->   [Application Data]
```

TLS 1.3 (incorrect key share)

```
ClientHello
+ key_share              -------->
                         <--------   HelloRetryRequest
ClientHello
+ key_share              -------->                 ...
```

Handshake State Machine

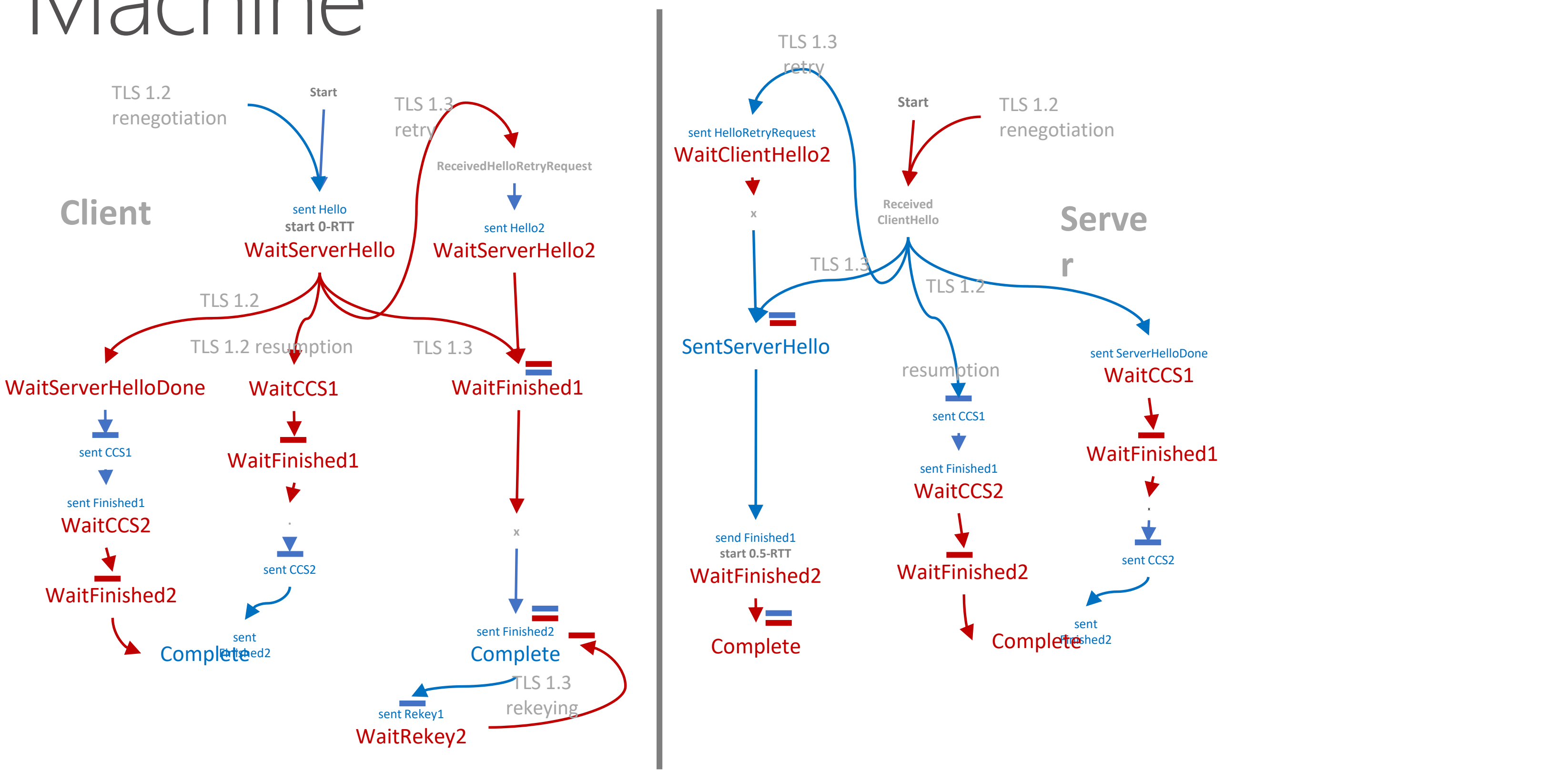
output

input

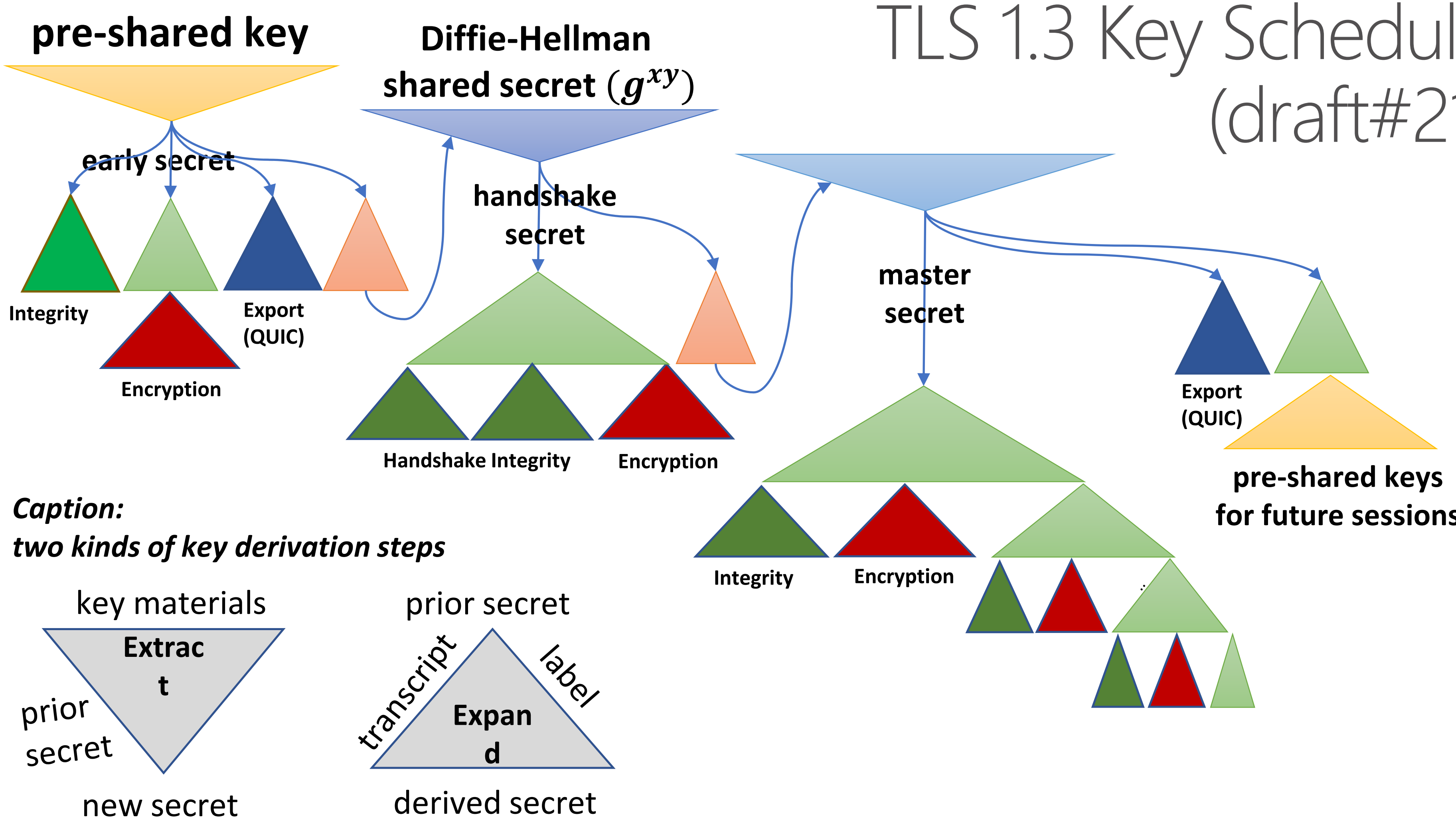
key changes

boxes represent successive flights

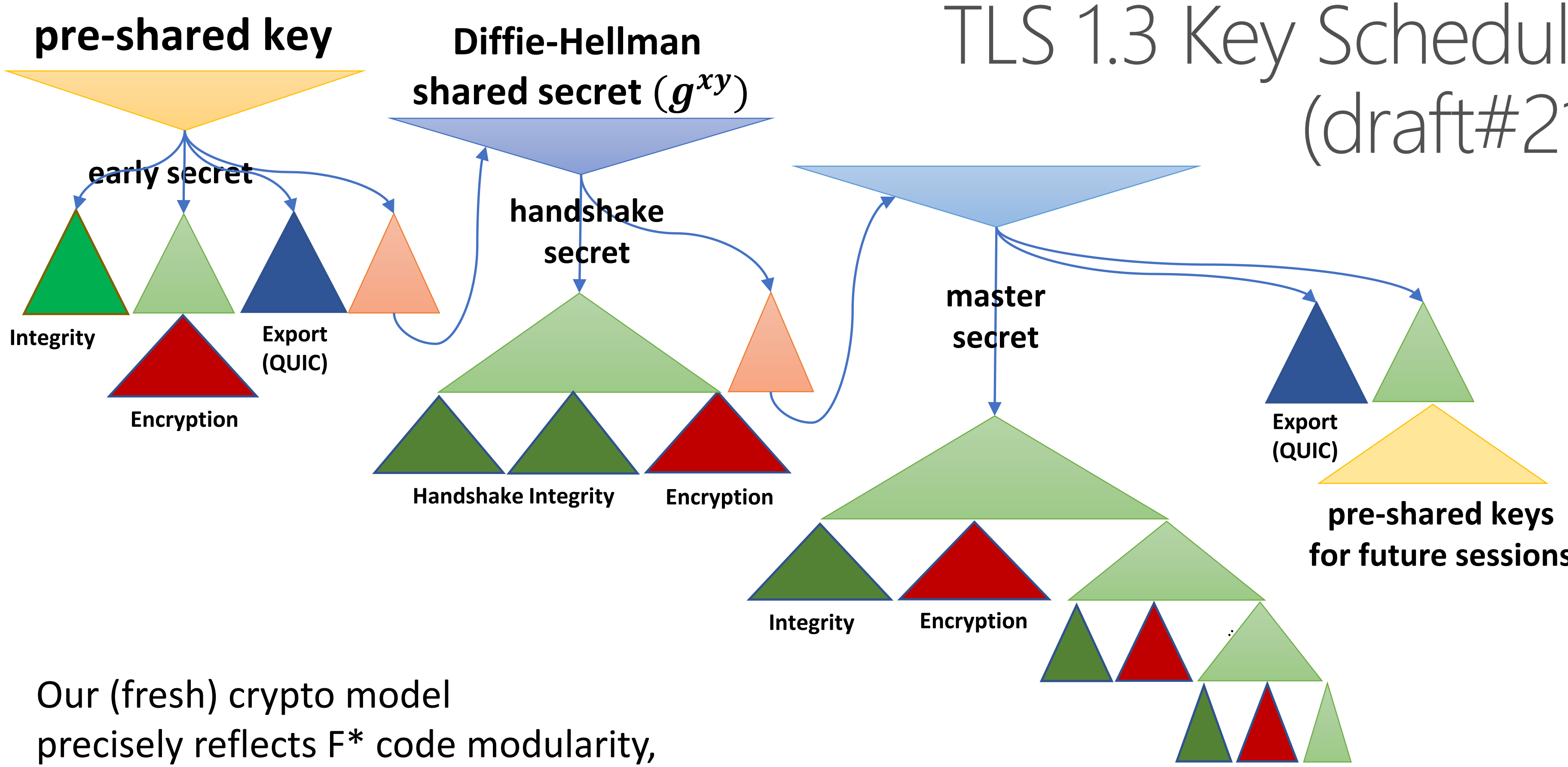
gray states need not be represented



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.

Exercise: RSA in F*

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

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

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

1. Simple specification
2. Fast exponentiation: square & multiply
3. Blinded implementation
   (for side-channel resistance)

Exercise: RSA in F*

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

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

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

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

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

$$Sign(m, (N, d)) = s' r^{-1} \ [N]$$

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