Verified Implementations for Real-World Cryptographic Protocols

Karthikeyan Bhargavan

Inria

+ many co-authors at Inria, MSR, CMU, Stuttgart, …

Collège de France, March 31 2022

LCL - Banque et assurance

lcl.fr

Incognito

LCL BANQUE ET ASSURANCE

Ma vie. Ma ville. Ma banque.

Vous êtes un particulier | Nous contacter | Devenir client | Mon espace

COMPTES ET CARTES | EPARGNER | EMPRUNTER | S'ASSURER | SIMULATEURS ET DEVIS | Ma Vie | Le Mag | Découvrir LCL

ALERTE FRAUDE : Nous constatons une vague d'appels téléphoniques frauduleux, pour en savoir plus cliquez ici.

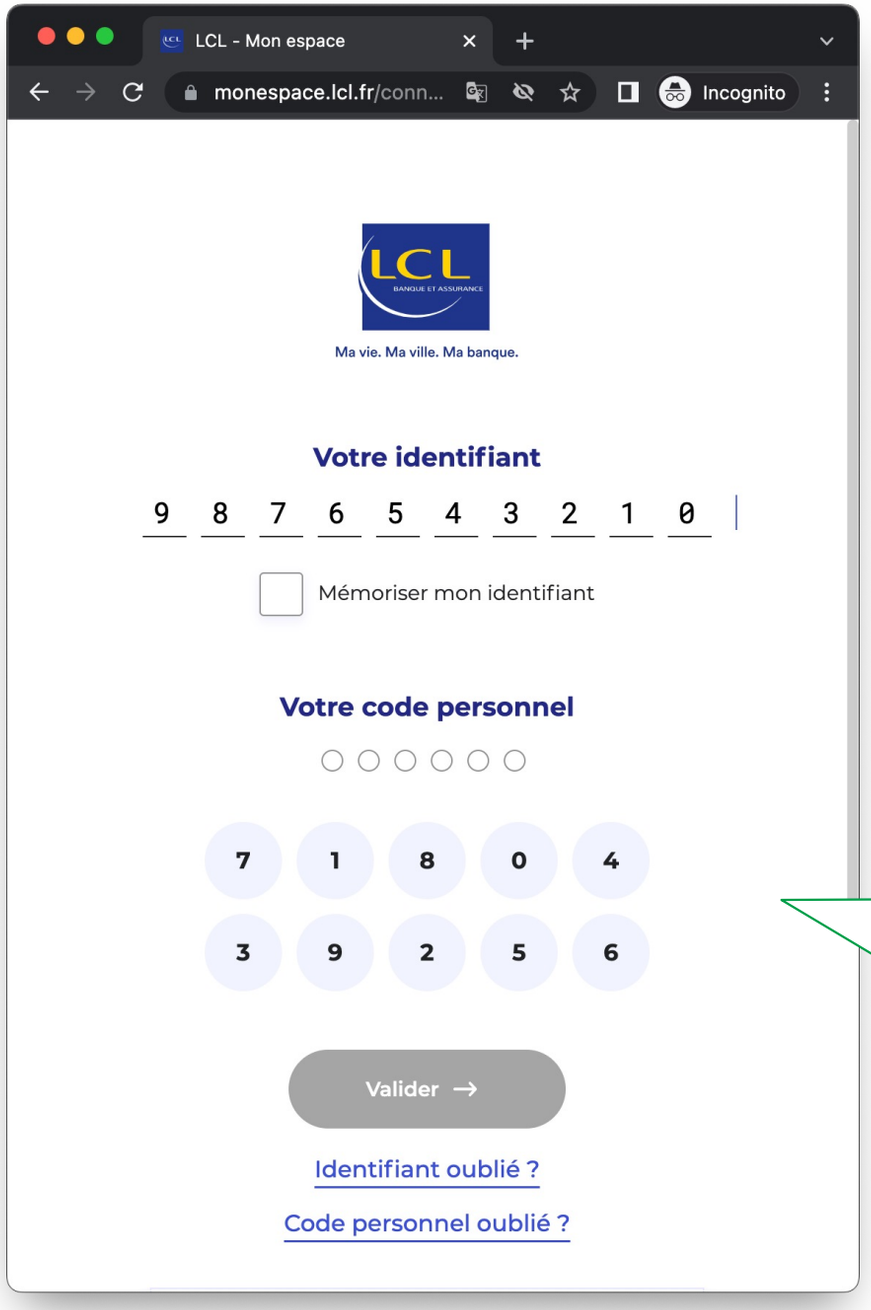
LCL BANQUE ET ASSURANCES

ÉLU SERVICE CLIENT DE L'ANNÉE 2022

40€ offerts pour toute première ouverture de compte !

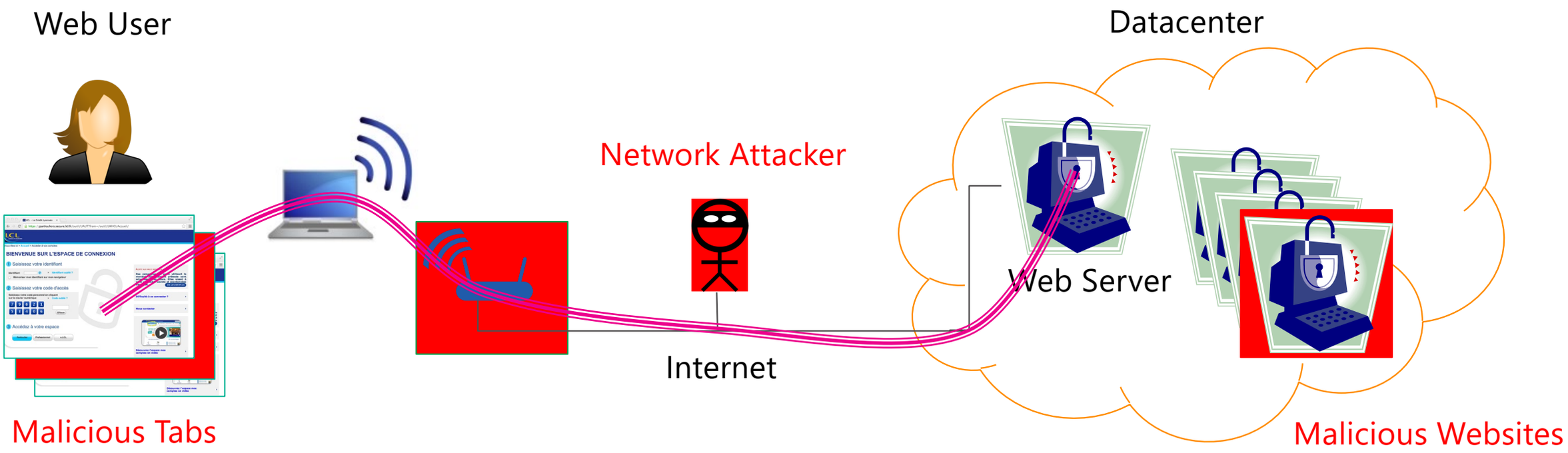
Profitez-en jusqu'au 31 mars ! →

Parlons de vos besoins

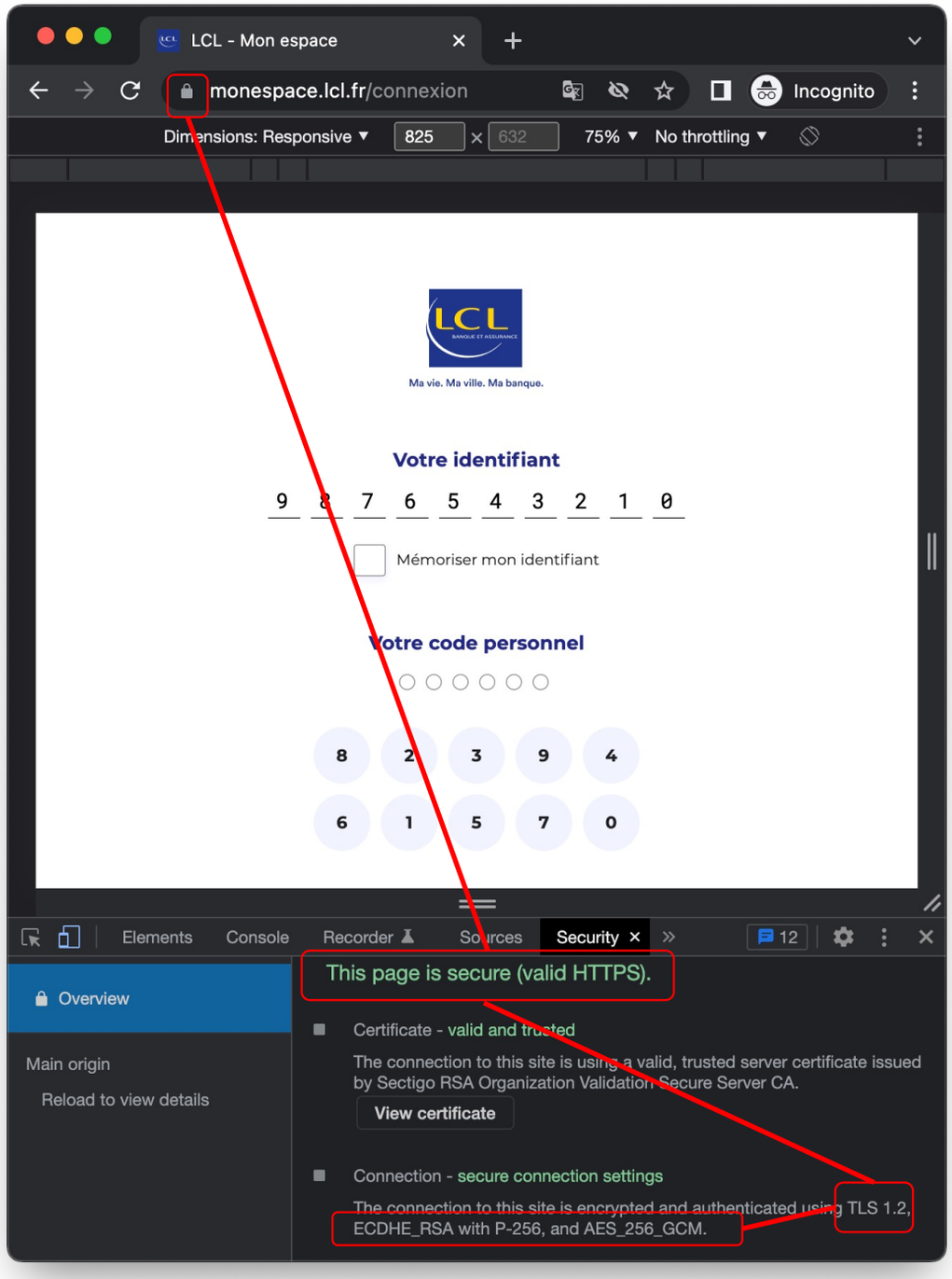


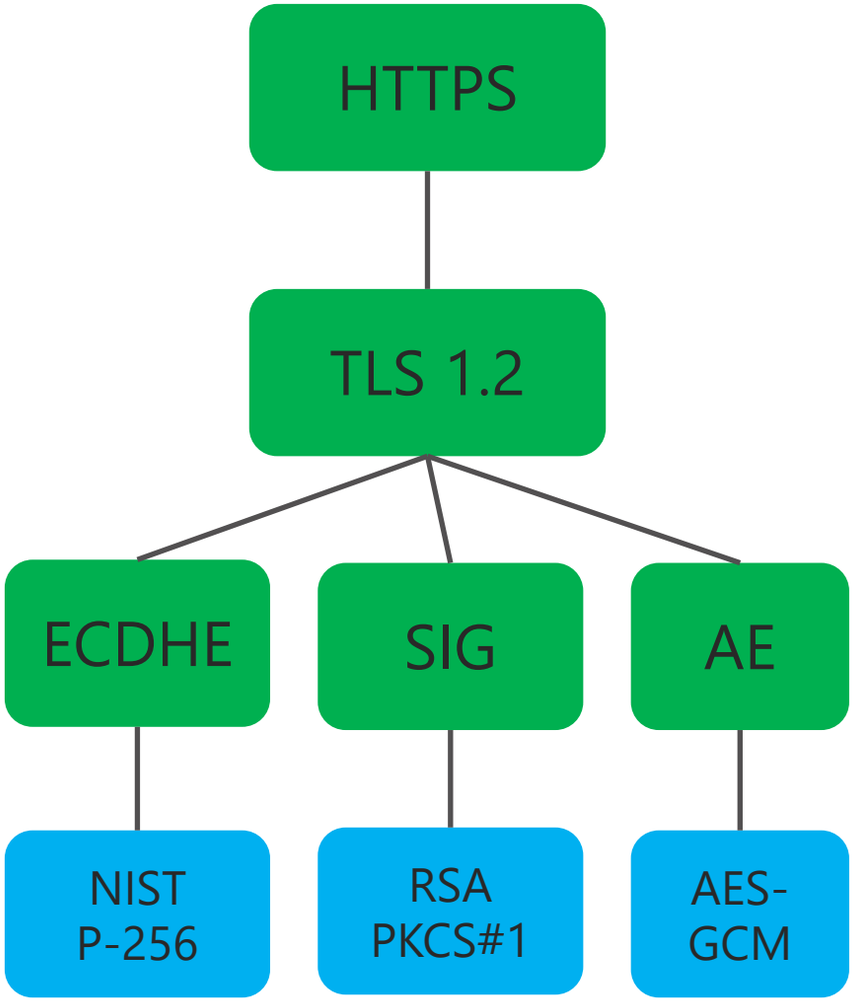
Is it safe to enter my pincode?

Secure Channels over Insecure Networks

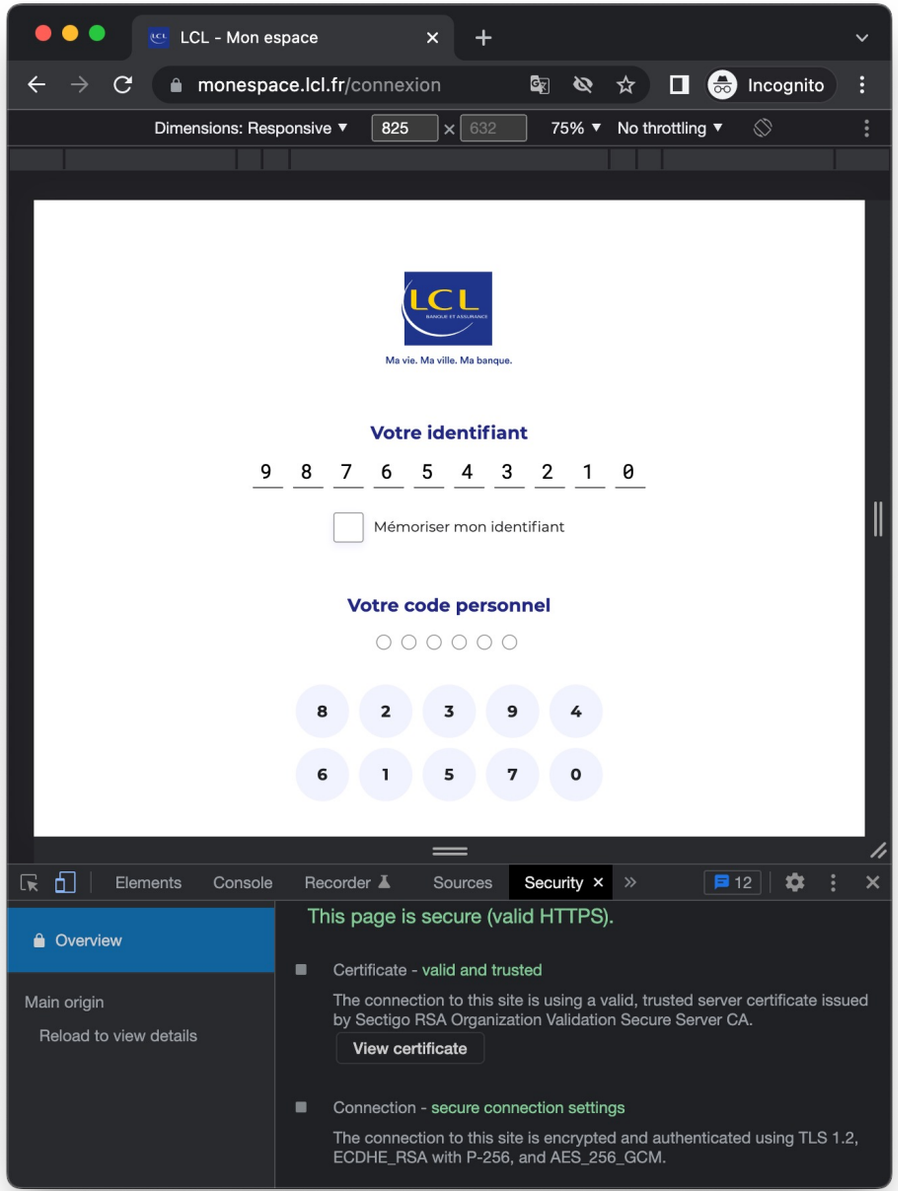


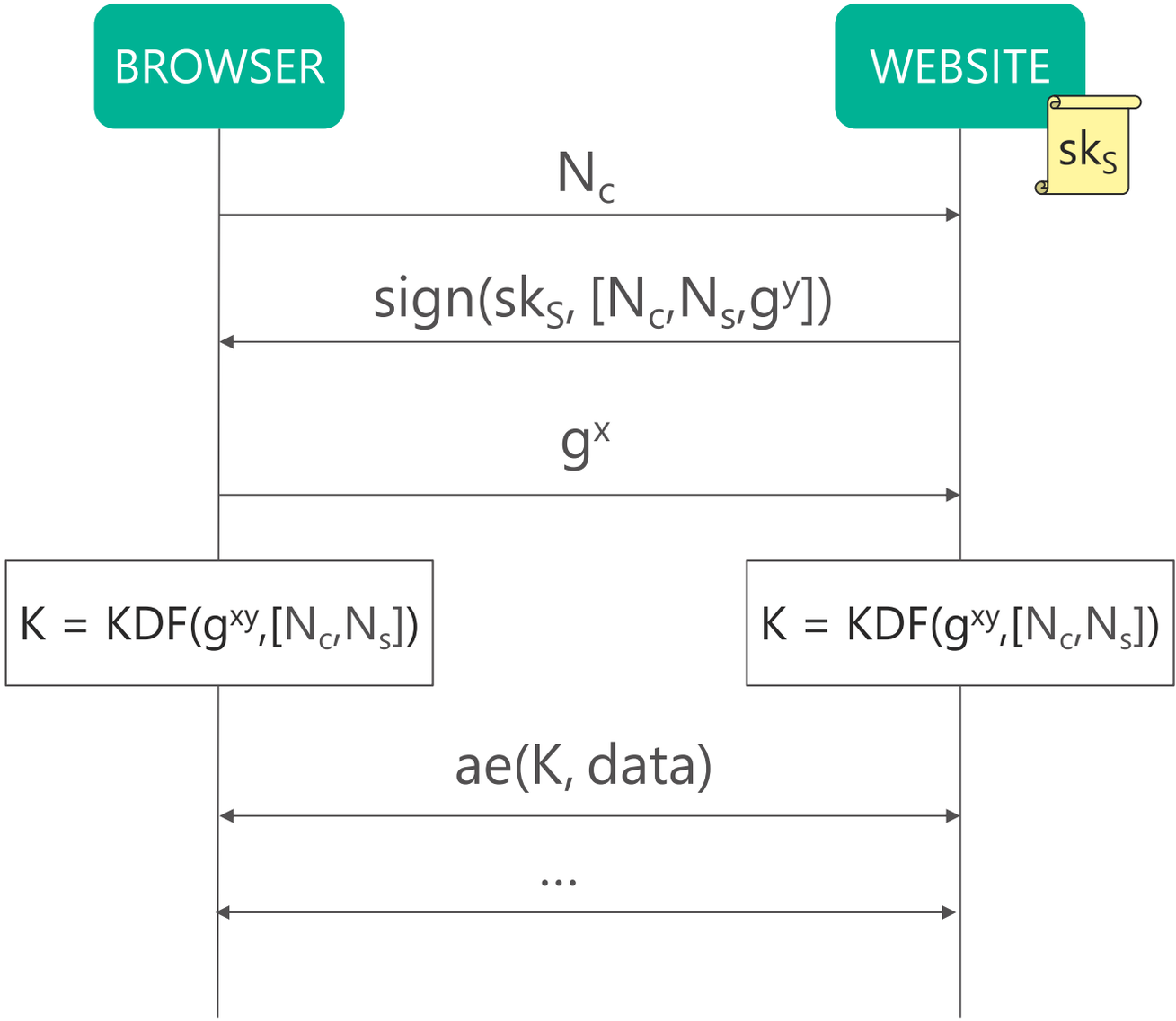
- We need a secure (authentic, confidential) channel
- Security against powerful attackers who may
  - Read all data sent on the network
  - Tamper with the contents of messages
  - Impersonate a user or a server


LCL - Mon espace
monespace.lcl.fr/connexion
Incognito
Dimensions: Responsive
825
632
75%
No throttling
Ma vie. Ma ville. Ma banque.
Votre identifiant
9 8 7 6 5 4 3 2 1 0
Mémoriser mon identifiant
Votre code personnel
8 2 3 9 4
6 1 5 7 0
Elements
Console
Recorder
Sources
Security
12
Overview
Main origin
Reload to view details
This page is secure (valid HTTPS).
Certificate - valid and trusted
The connection to this site is using a valid, trusted server certificate issued by Sectigo RSA Organization Validation Secure Server CA.
View certificate
Connection - secure connection settings
The connection to this site is encrypted and authenticated using TLS 1.2, ECDHE_RSA with P-256, and AES_256_GCM.


HTTPS
TLS 1.2
ECDHE
SIG
AE
NIST P-256
RSA PKCS#1
AES-GCM

Secure Channels in TLS [1994-2008]



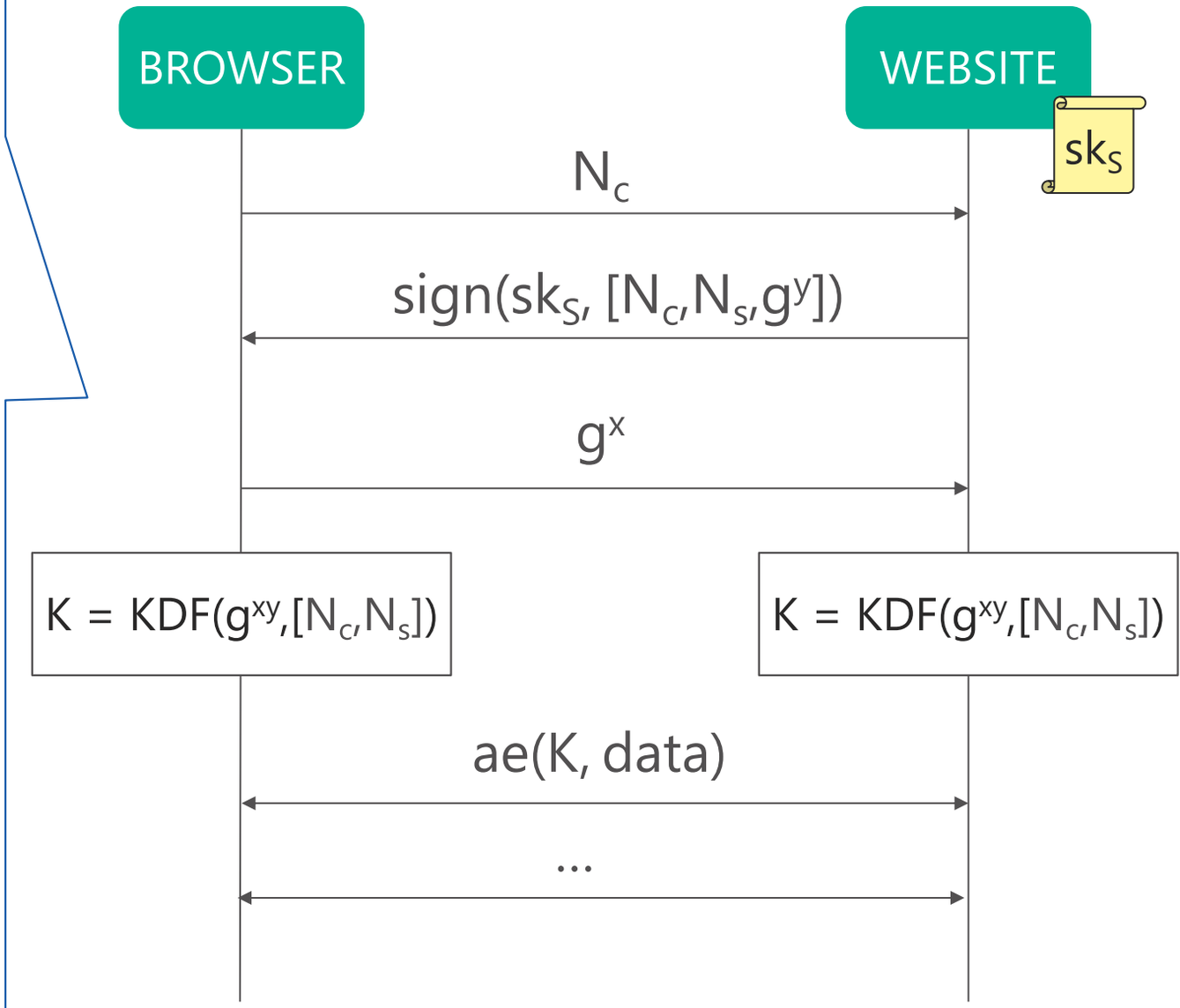


Secure Channels in TLS [1994-2008]

Classic Two-Stage Protocol

1. **Authenticated Key Exchange**
   Both parties compute a shared secret key K
2. **Authenticated Encryption**
   Both parties exchange streams of data encrypted under K

- Many variants (IPsec, SSH, TLS)
- Many security models & proofs
- So, is this a solved problem?



Many Attacks on TLS Deployments [2011-16]

• BEAST	CBC predictable IVs	[Sep'11]
• CRIME	Compression before Encryption	[Sep'12]
• RC4	Keystream biases	[Mar'13]
• Lucky 13	MAC-Encode-Encrypt CBC	[May'13]
• 3Shake	Insecure resumption	[Apr'14]
• POODLE	SSLv3 MAC-Encode-Encrypt	[Dec'14]
• FREAK	Export-grade 512-bit RSA	[Mar'15]
• LOGJAM	Export-grade 512-bit DH	[May'15]
• SLOTH	RSA-MD5 signatures	[Jan'16]
• DROWN	SSLv2 RSA-PKCS#1v1.5	[Mar'16]
• SWEET32	3DES collisions	[May'16]

What goes wrong in TLS Deployments?

Crypto Weaknesses

- RC4, 3DES, MD5, PKCS#1 v1.5

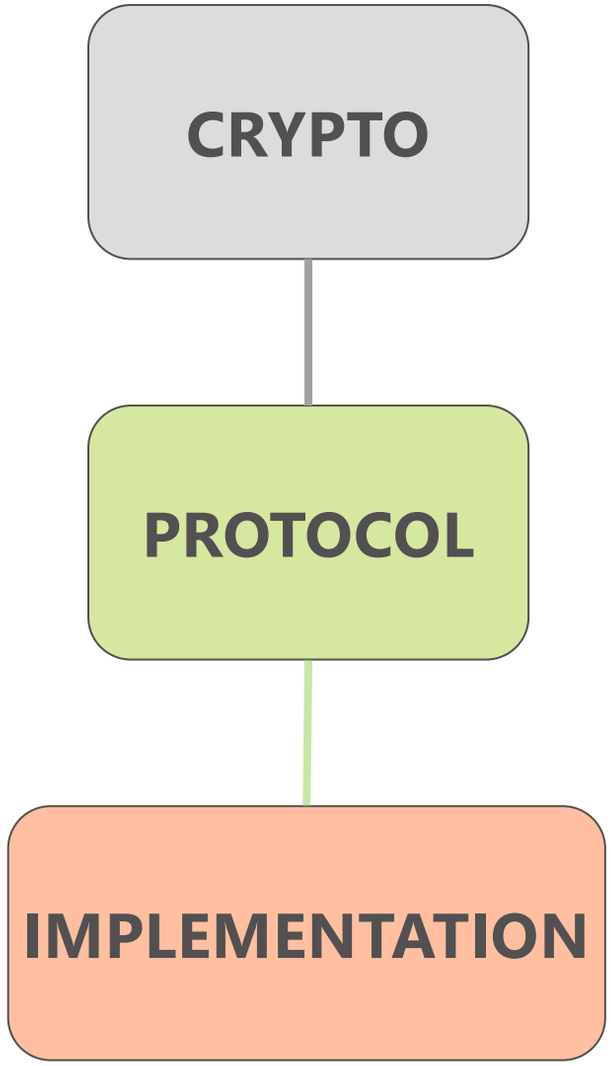
Protocol Design Flaws

- Downgrade attacks, Transcript collisions

Implementation Bugs

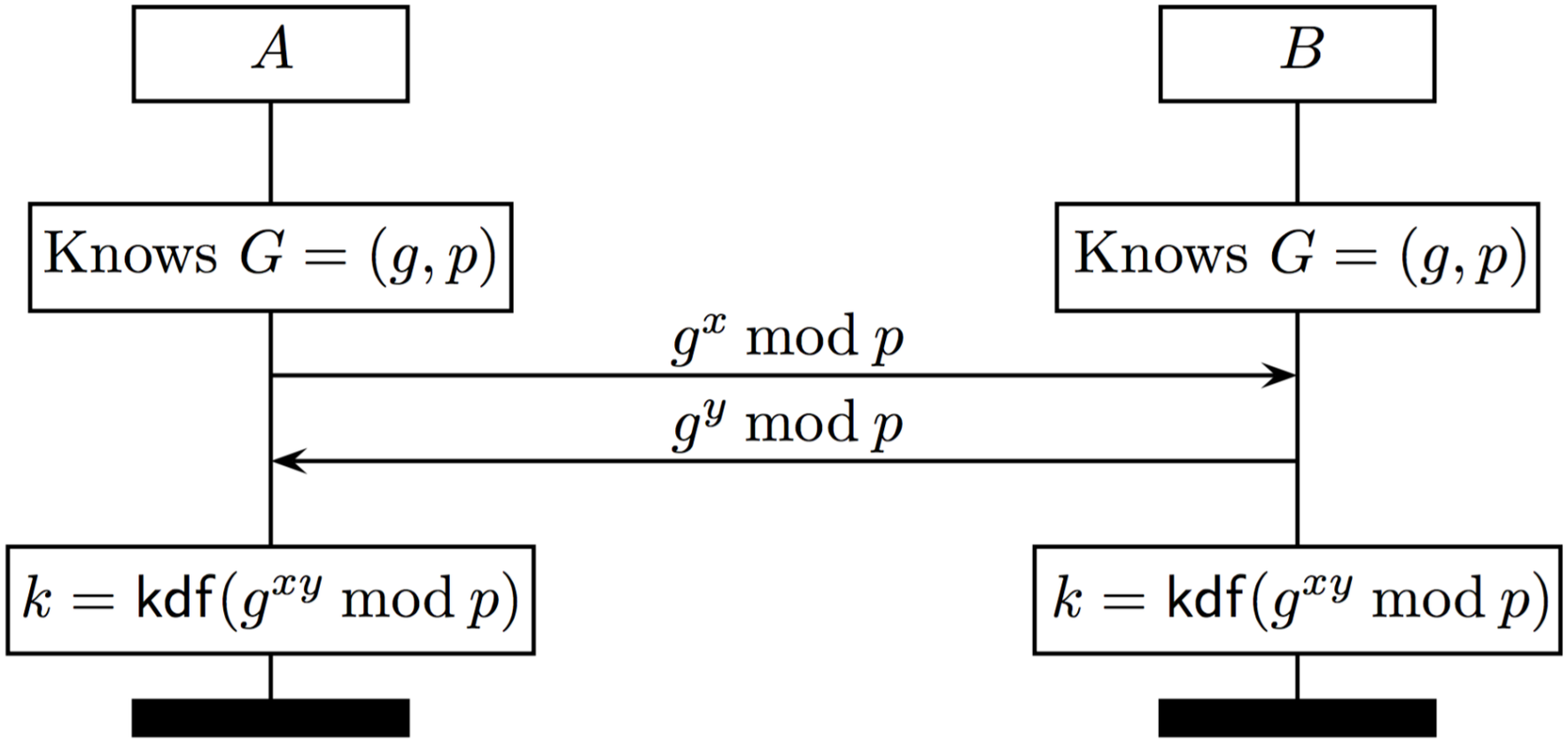
- State machine bugs, Heartbleed

Often, a mix of all of the above!



Crypto Weaknesses

Diffie-Hellman Key Exchange



Diffie-Hellman Assumption

$$g^{xy} \mod p$$

$0 < x,y < p$
g fixed

Large Prime
(e.g. 2048 bits)

Security Assumption: An attacker who does not know x or y cannot compute $g^{xy} \bmod p$

Weak Diffie-Hellman Groups

If the prime p is too small,
an attacker can compute the discrete log:

$$y = \log(g^y \bmod p)$$

and hence compute the session key: $g^{xy} \bmod p$

Current discrete log computation records:

- [Joux et al. 2005] 431-bit prime
- [Kleinjung et al. 2007] 530-bit prime
- [Bouvier et al. 2014] 596-bit prime
- [Kleinjung et al. 2017] 768-bit prime
- [Boudot et al. 2019] 795-bit prime

Real-World Diffie-Hellman Groups

Internet-wide scan of HTTPS servers (2015)

- 14.3M hosts, 24% support DHE
- 70,000 distinct groups *(p,g)*

Many small-sized prime groups used for TLS

- 84% (2.9M) servers use 1024-bit primes
- 2.6% (90K) servers use 768-bit primes
- 0.0008% (2.6K) servers use 512-bit primes

Many servers support both strong and weak groups.

Protecting Protocols from Weak Crypto

Many deployed crypto algorithms are now considered weak

- RC4, MD5, 3DES, RSA-PKCS#1v1.5

The need for **backwards compatibility**

- Many systems cannot be updated frequently
- Need to continue support for legacy clients/servers

The benefits of **cryptographic agility**

- Gracefully transition from one algorithm to another
- Can already start supporting Post-Quantum algorithms

Protecting Protocols from Weak Crypto

Prove the security of protocols under weak assumptions

- Do you really need a collision-resistant hash function?
- Do you really need an IND-CCA secure encryption algorithm?

Analyze protocols that support both strong and weak crypto

- Prove security for connections that use strong crypto
- Show that strong crypto cannot be bypassed using weak crypto

Analyzing agile protocols by hand is too hard

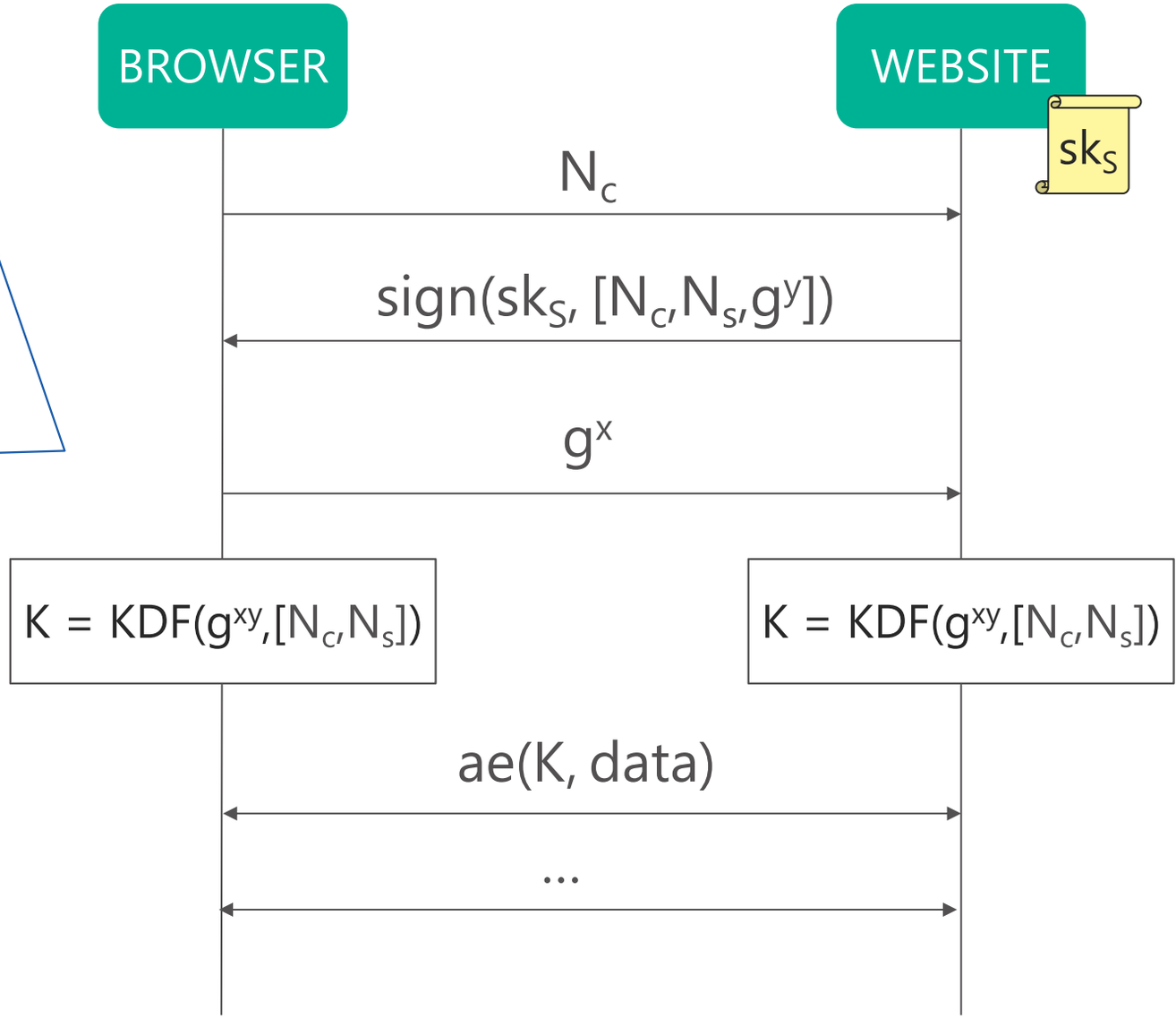
- Large models, subtle assumptions and goals
- Need mechanized protocol verification tools

Protocol Design Flaws

Composing Cryptographic Constructions

Each crypto protocol composes a set of crypto constructions to achieve some target security goals

- TLS = DH + Sign + KDF + AE
- Each crypto algorithm may be individually strong, but they may not collectively achieve the desired security goal



Composing Sub-Protocols

Sequential or Vertical Composition

- Values generated by one protocol are used in the next
- e.g. Authenticated Key Exchange + Authenticated Encryption

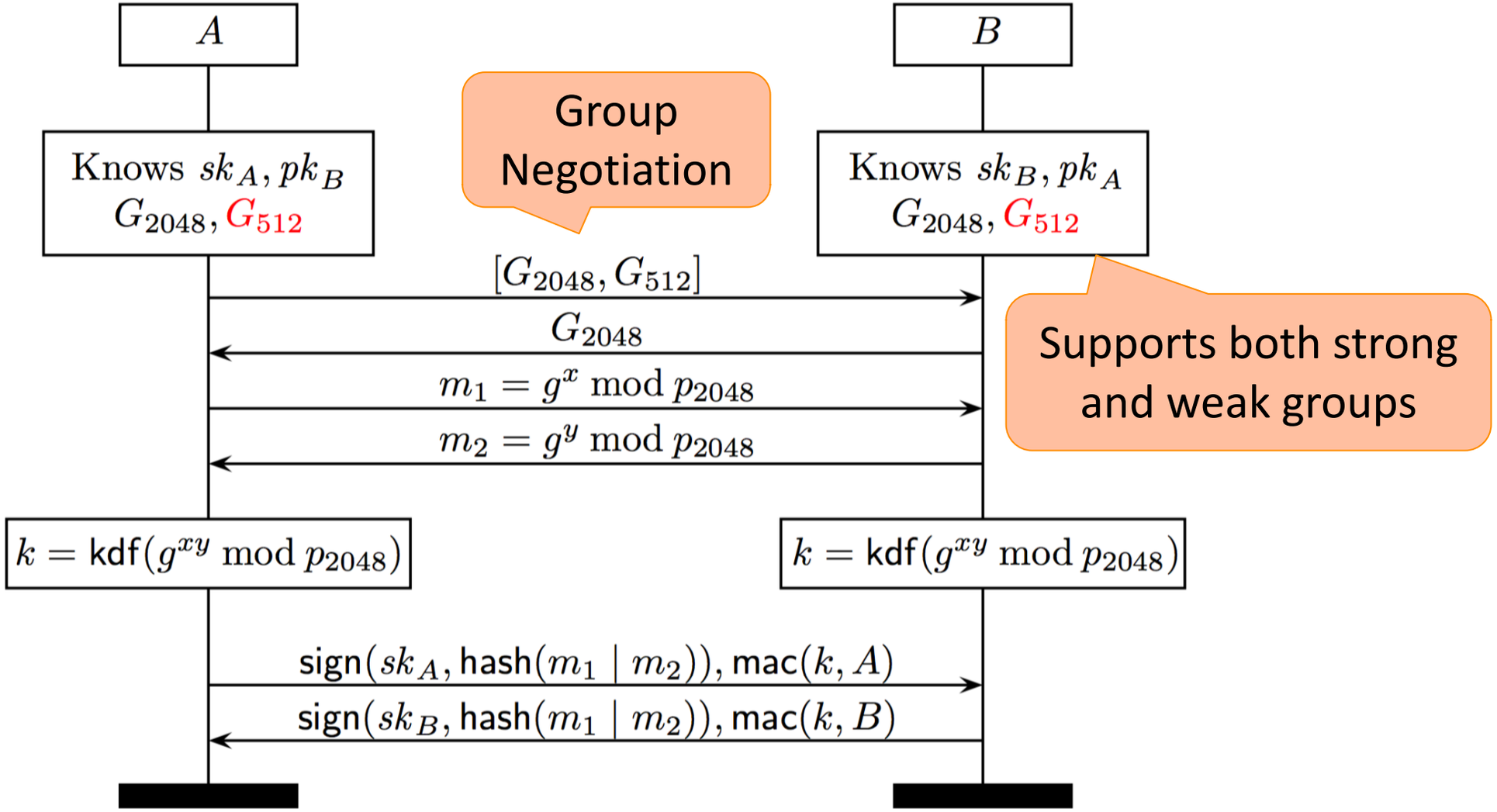
Protocols with Algorithmic Agility

- Support for multiple algorithms within a single protocol
- e.g. allow weak and strong Diffie-Hellman groups

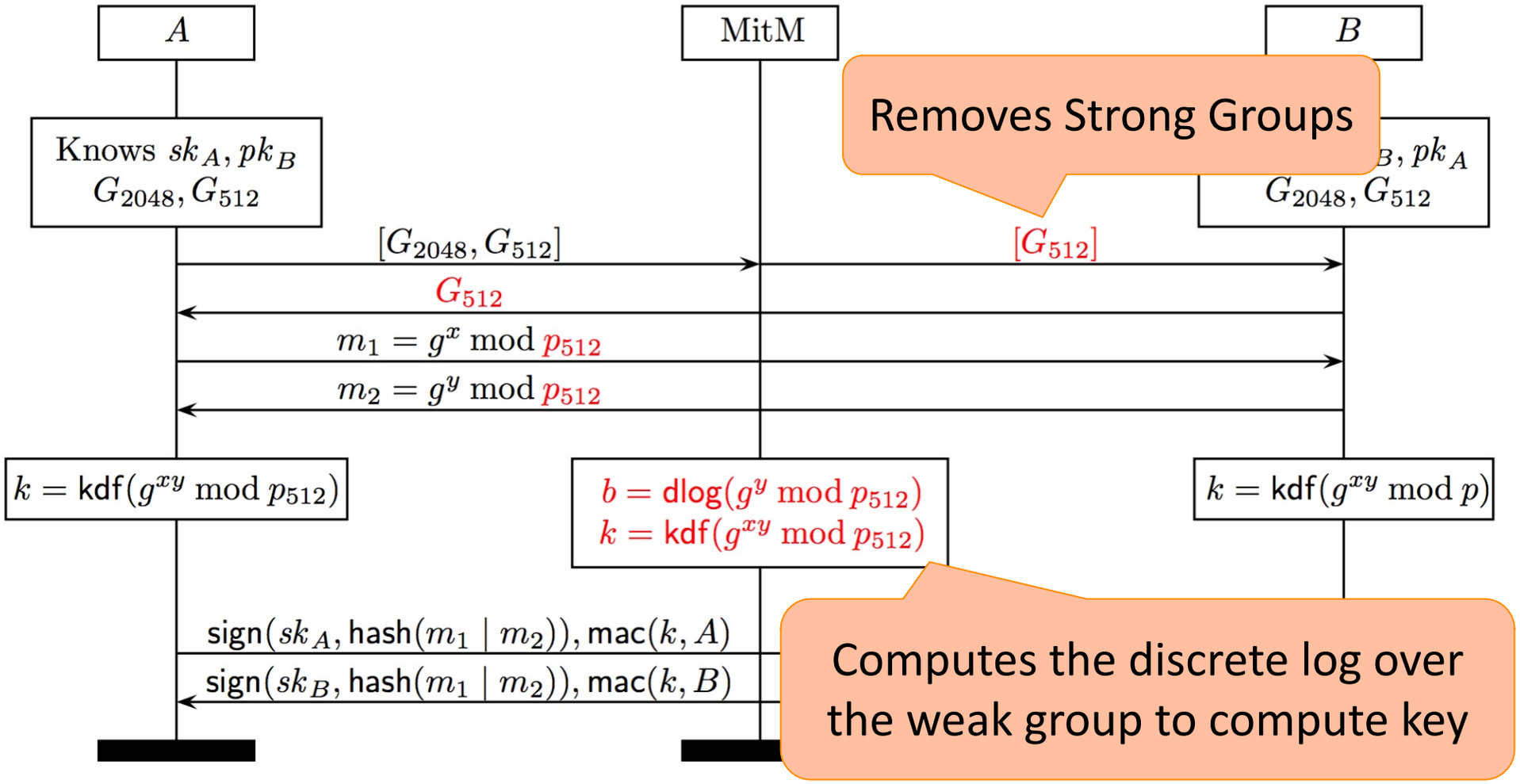
Parallel or Multi-mode Composition

- Many protocol flows to choose from
- Different sessions may choose a different modes

Agility: Diffie-Hellman Group Negotiation



Group Downgrade Attack



What went wrong?

Logjam Attack [2015]

- Cryptographic weakness: Weak Diffie-Hellman Groups
- Logical protocol flaw: Downgrade from Strong to Weak Group

Many other examples of downgrade+crypto attacks

- FREAK, SLOTH, DROWN, …

These attacks only appear when analyzing complex composite protocol deployments

Implementation Bugs

Bugs in Protocol Implementations

Bugs when implementing cryptographic algorithms

- Functional correctness bugs, Side-channel leaks, …
- e.g. Lucky13, Bleichenbacher, see OpenSSL CVEs

Bugs when parsing protocol messages and components

- Memory safety bugs (Heartbleed), Error propagation (Gotofail)
- X.509 certificate parsing errors (many CVEs)

Bugs in protocol state machine implementation (next)

- Allowing incorrect protocol flows (FREAK, SKIP)

Many possible protocol modes of TLS

Protocol versions

- TLS 1.2, TLS 1.1, TLS 1.0, SSLv3, SSLv2

Key exchanges

- ECDHE, FFDHE, RSA, PSK, …

Authentication modes

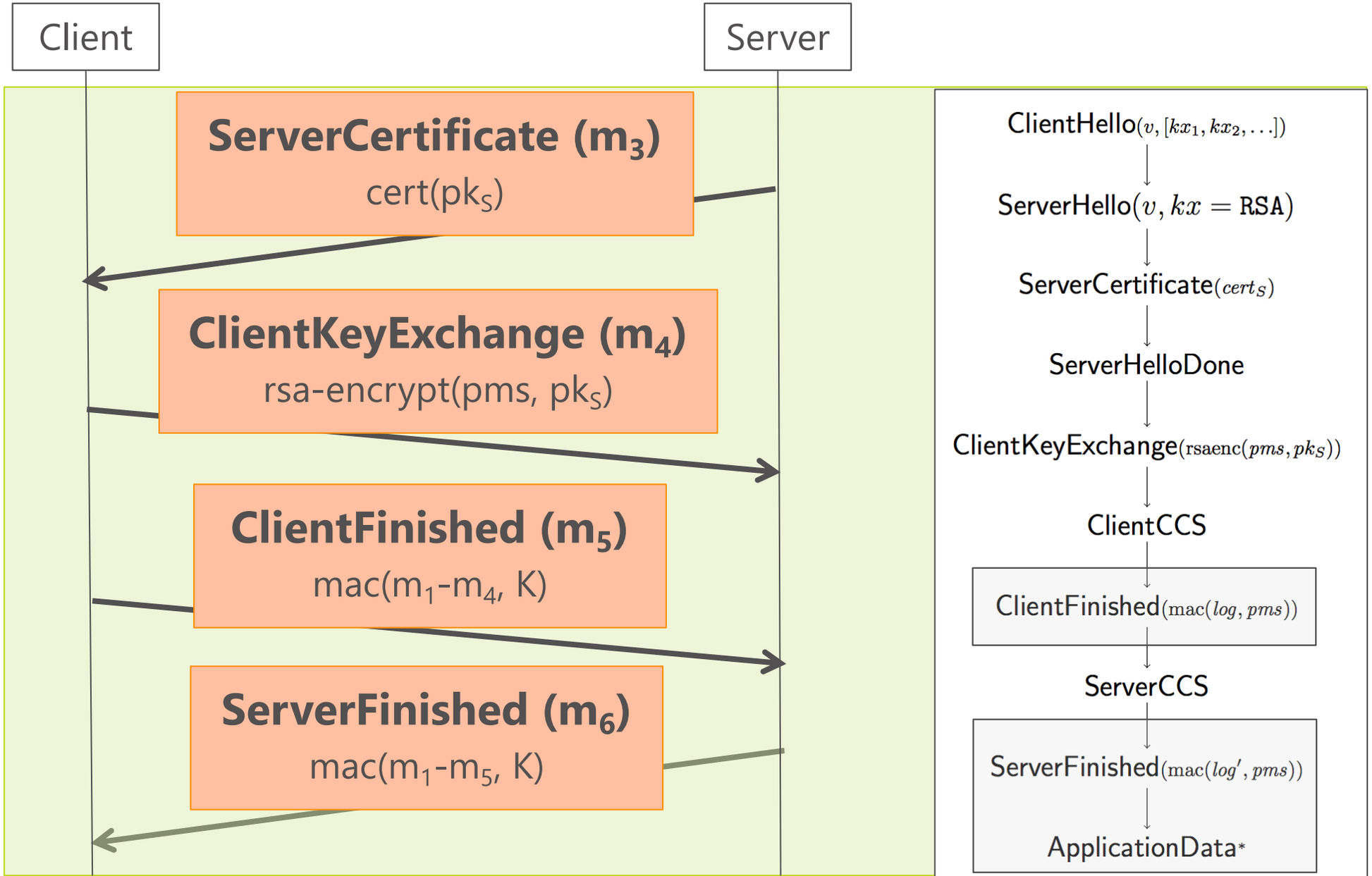
- ECDSA, RSA signatures, PSK,…

Authenticated Encryption Schemes

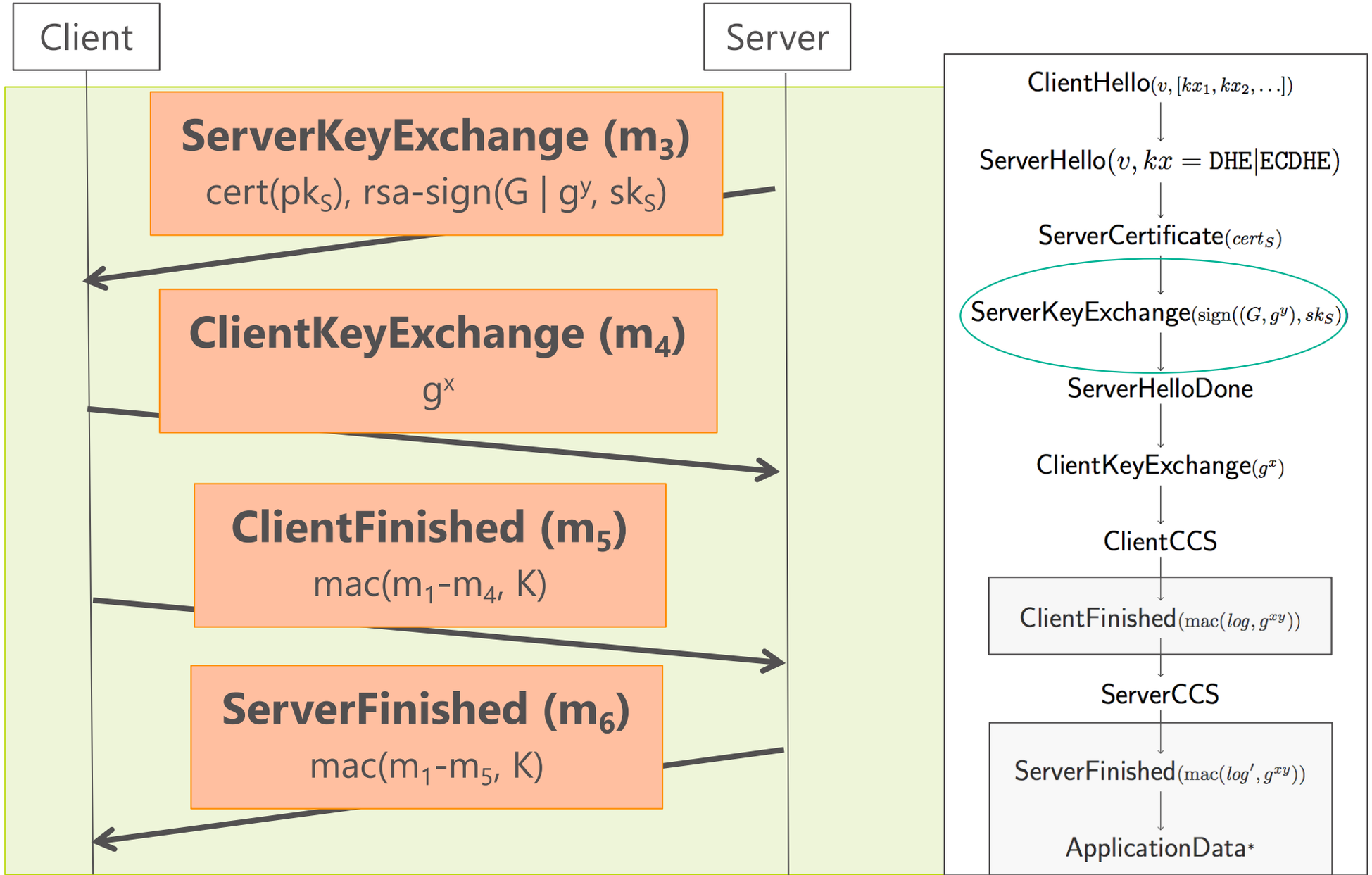
- AES-GCM, CBC MAC-Encode-Encrypt, RC4,…

100s of possible protocol combinations!

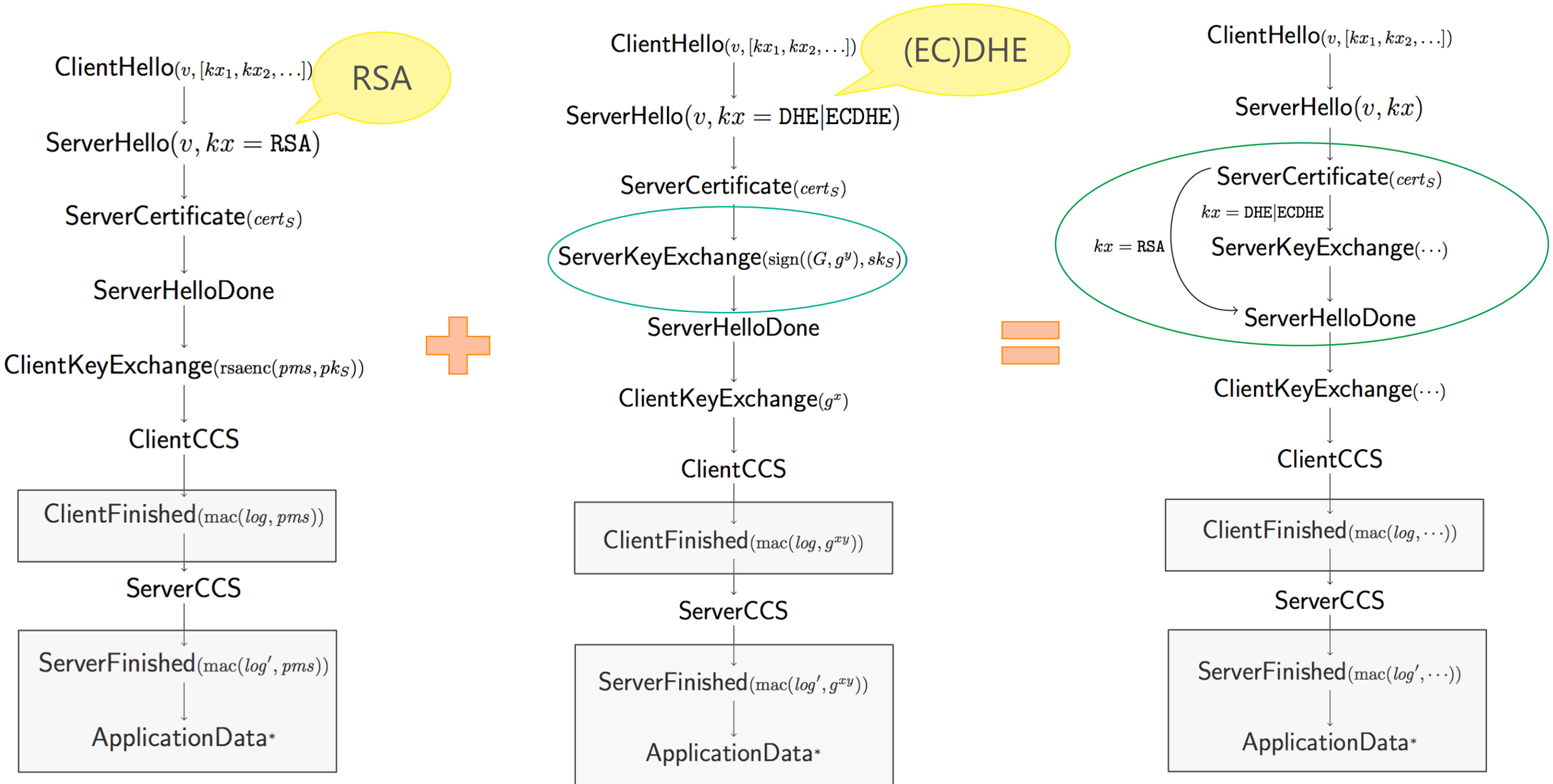
State Machine for TLS-RSA Key Exchange



State Machine for TLS-DHE Key Exchange



Composing Protocol State Machines



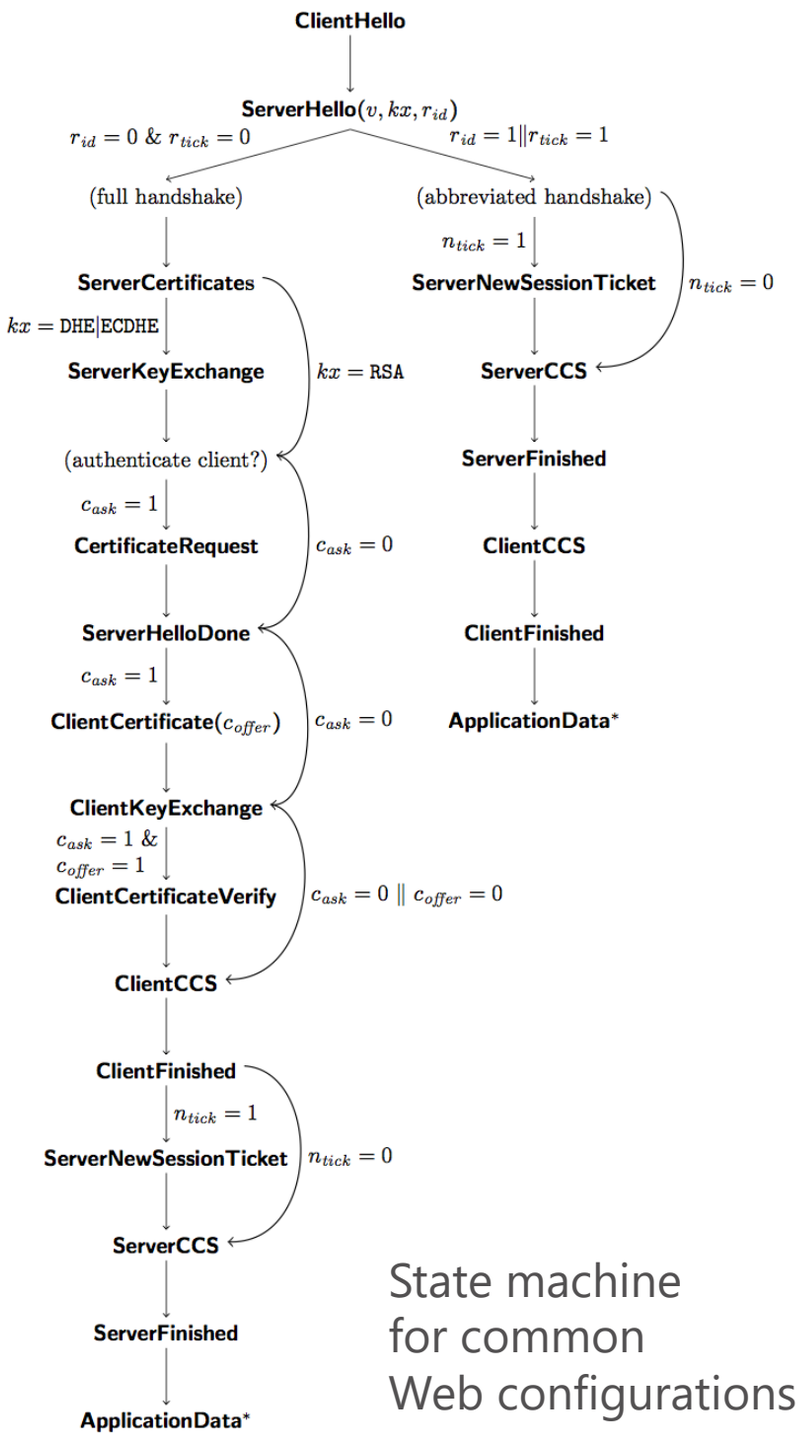
Commonly Deployed TLS State Machine

RSA + DHE + ECDHE

+ Session Resumption

+ Client Authentication

- Covers most features used on the Web

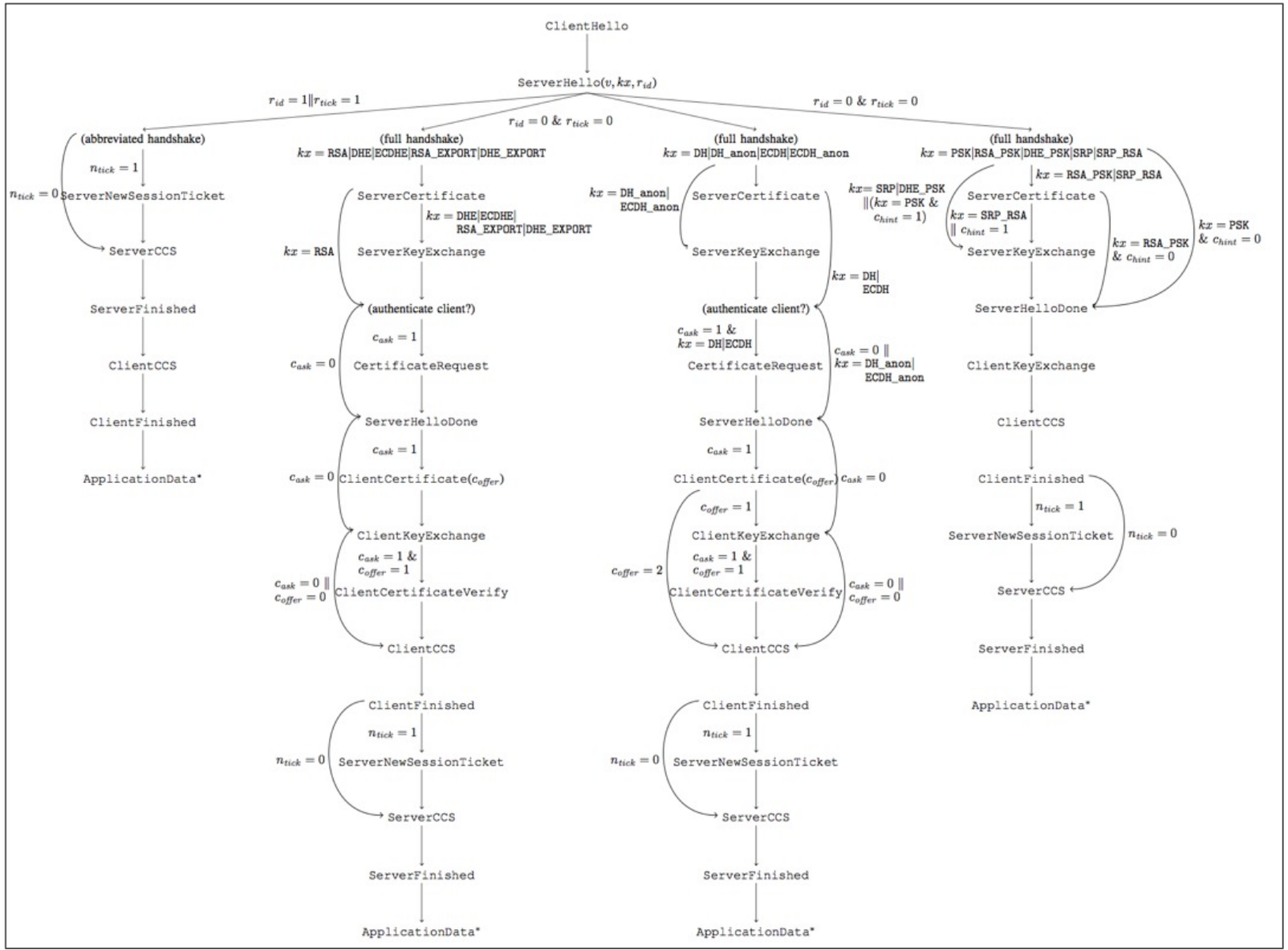


State machine for common Web configurations

Full SSL/TLS State Machine

+ Fixed_DH
+ DH_anon
+ PSK
+ SRP
+ Kerberos
+ *_EXPORT
+ …

These are all the ones implemented in OpenSSL



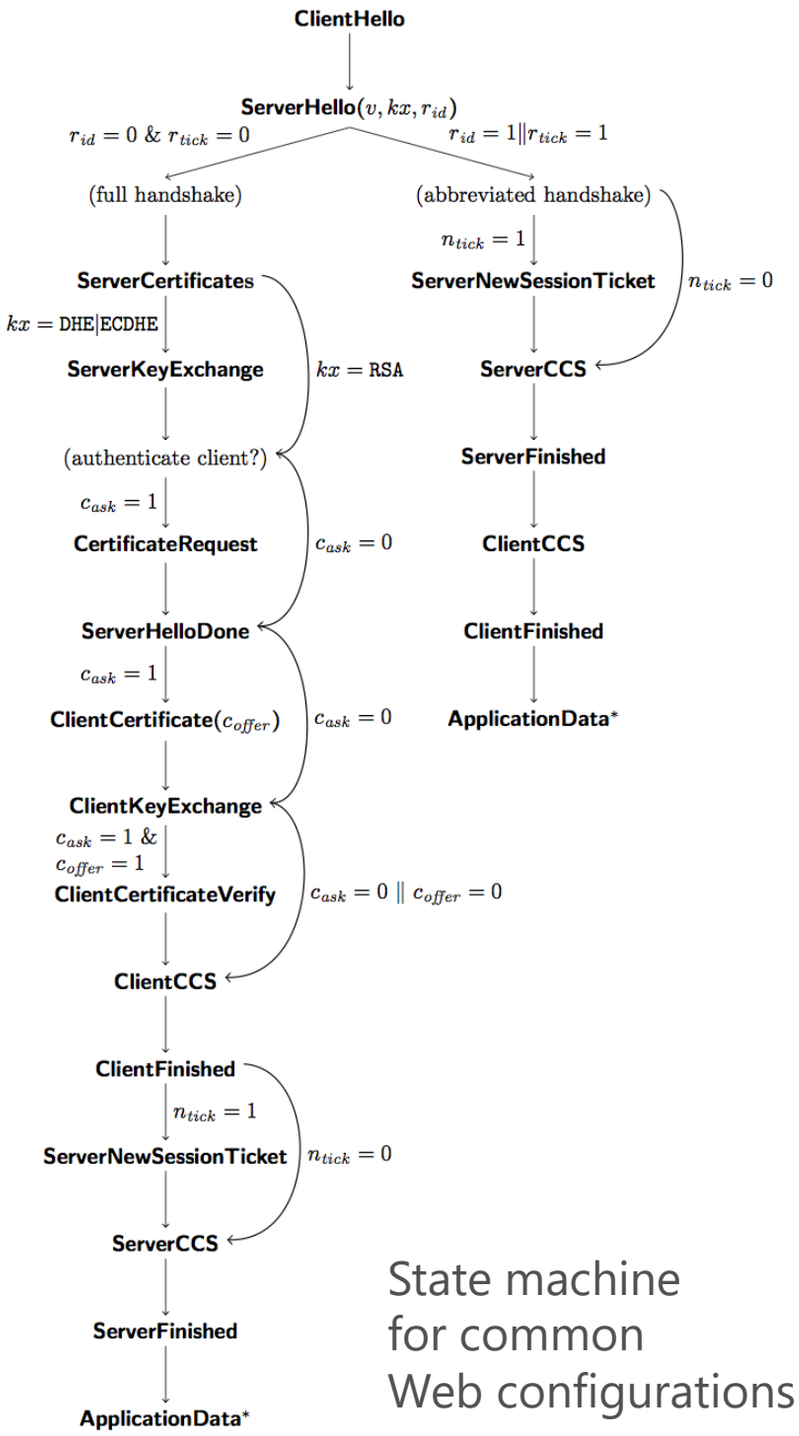
Testing TLS State Machines

Do popular TLS libraries conform to this state machine spec?

- Does OpenSSL? Firefox? Chrome? Safari? Java TLS? IoT devices?

We built a fuzzing framework

- FlexTLS, based on miTLS, a verified implementation of TLS
- Generates non-conforming traces from a *formal state machine spec*
- Tests open-source libraries

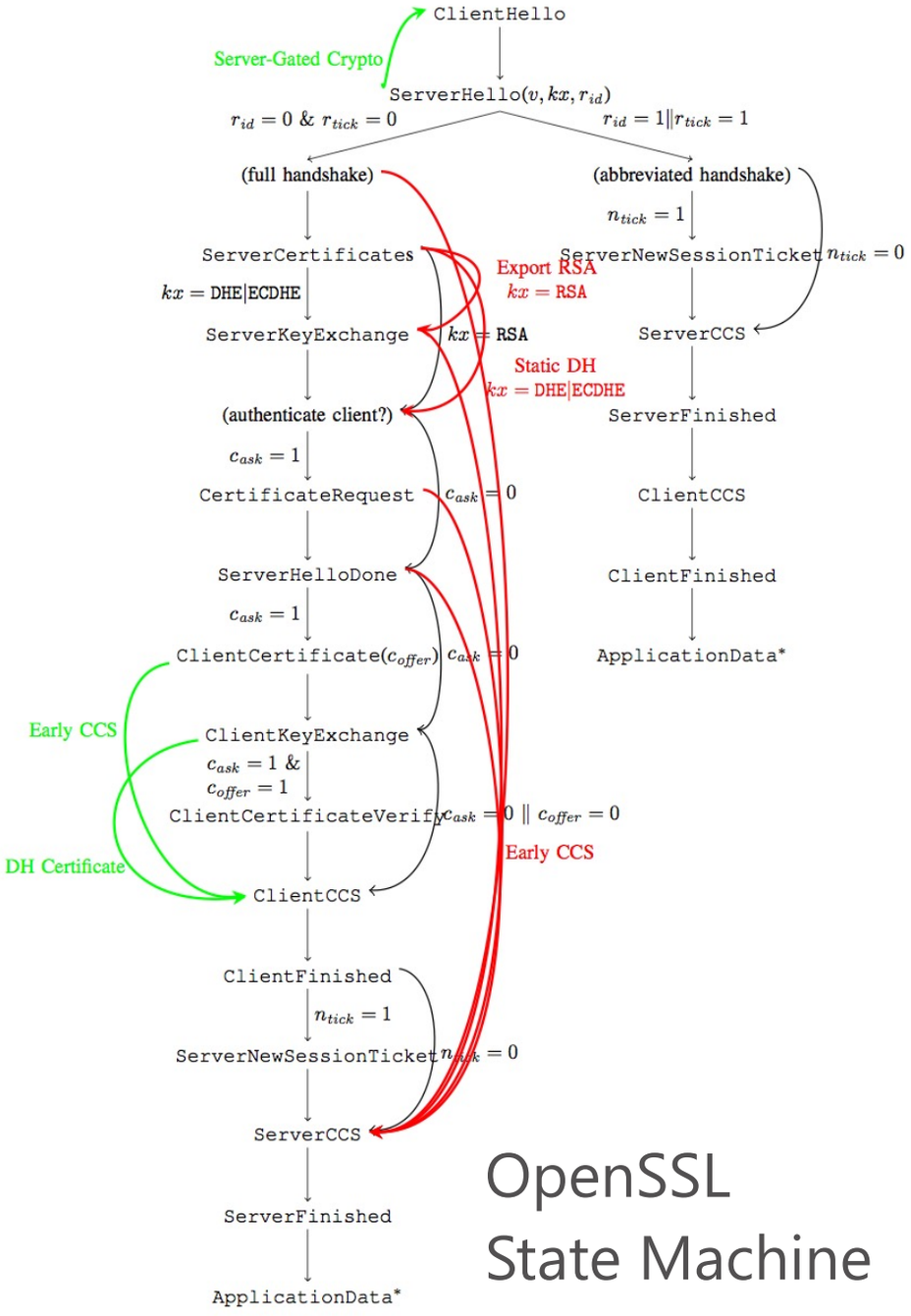


State machine for common Web configurations

Many, Many Bugs

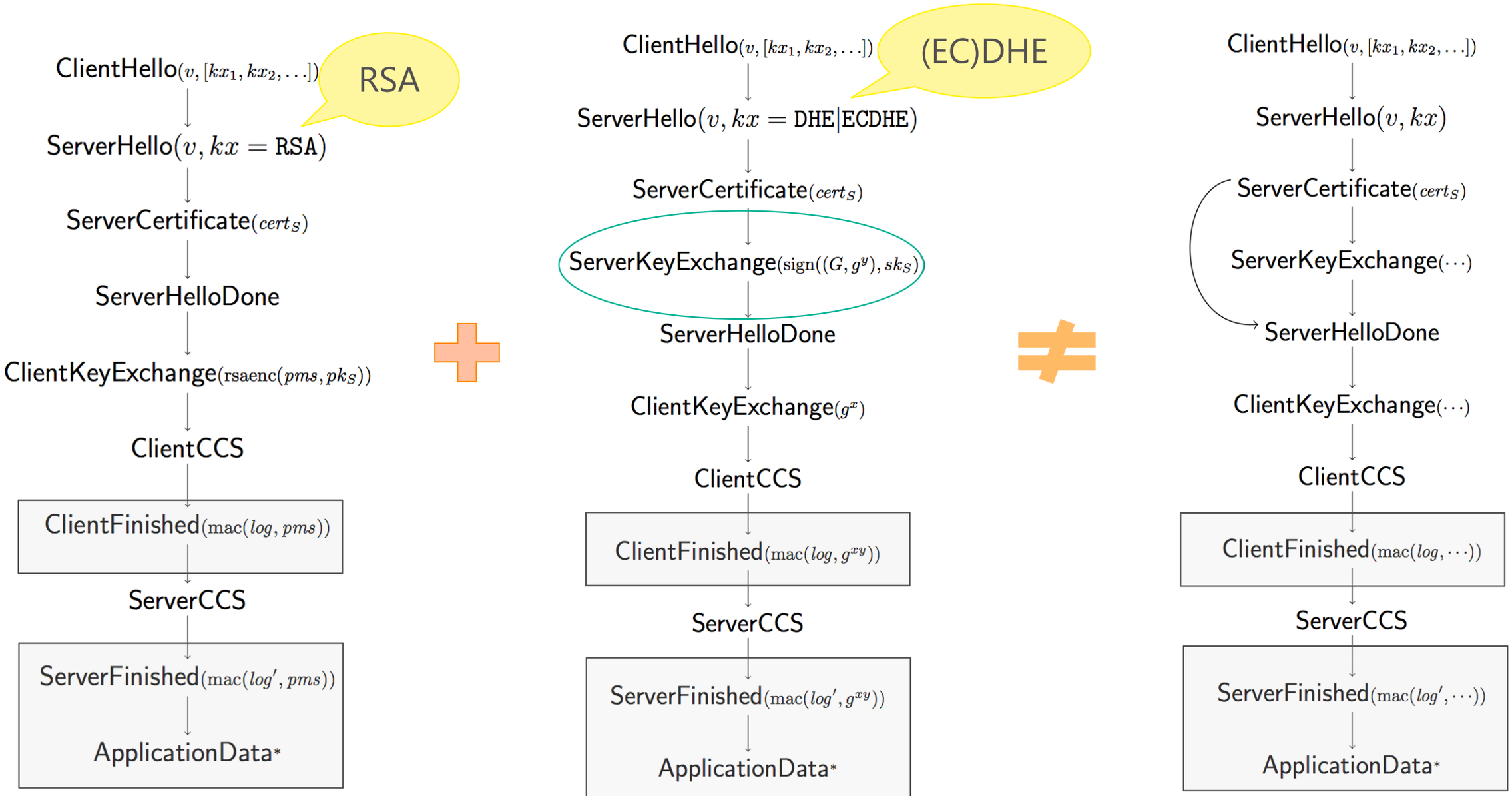
Unexpected state transitions in OpenSSL, NSS, Java, SecureTransport, …

- Required messages are allowed to be skipped
- Unexpected messages are allowed to be received
- CVEs for many libraries



OpenSSL State Machine

Incorrectly Composing State Machines



Incorrectly Composing State Machines

Follows Postel's robustness principle

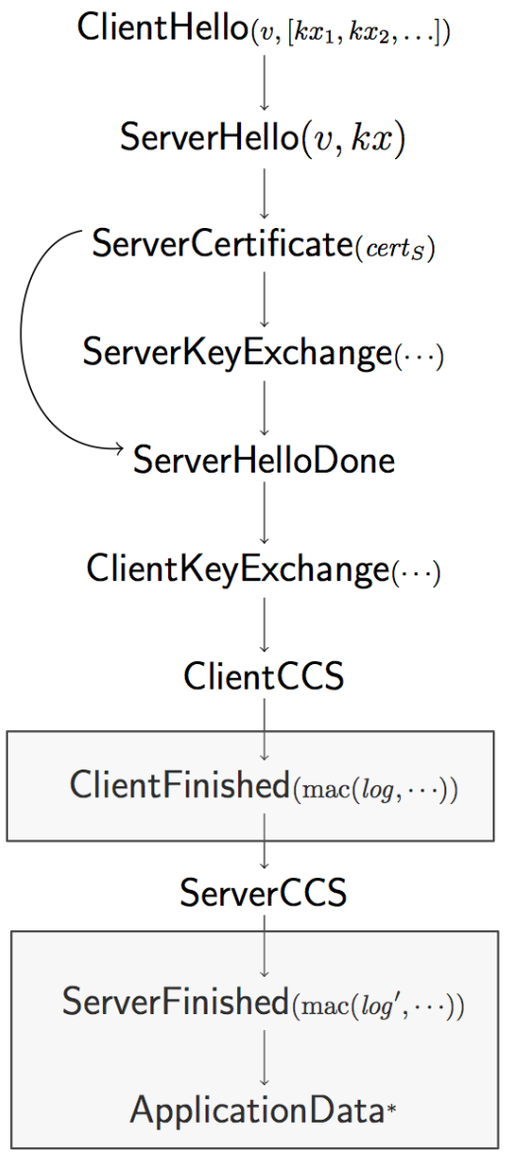
- "Be conservative in what you do,
  be liberal in what you accept from others"
  (BAD for security!)

Introduces unexpected cases at the client

- Server skips ServerKeyExchange in DHE
- Server sends ServerKeyExchange in RSA

Correct clients should reject these cases

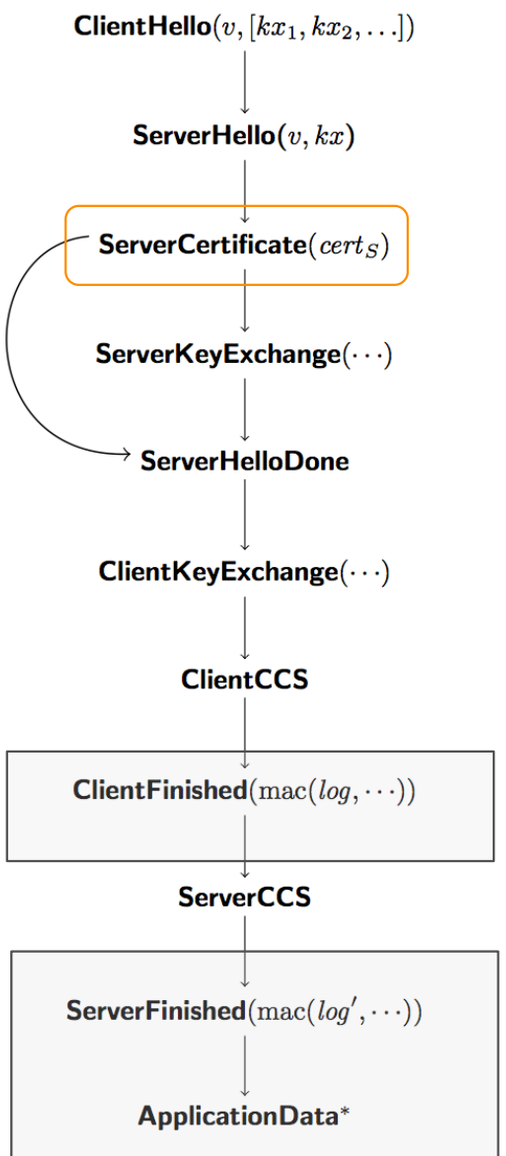
- Otherwise, they are not executing TLS anymore, and lose all its security guarantees



SKIP: skipping Messages to Java Clients

Network attacker impersonates api.paypal.com to a JSSE client

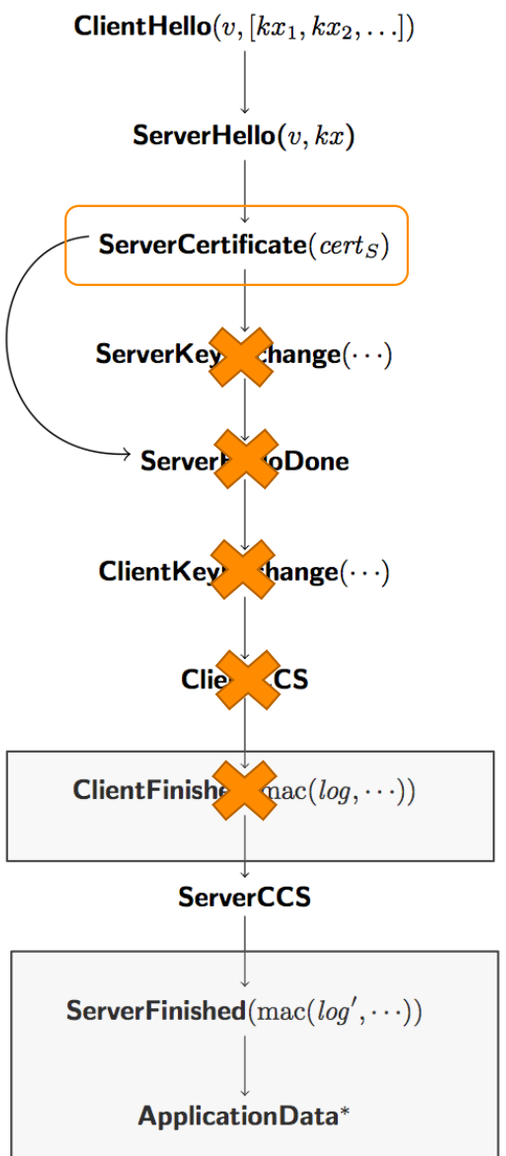
1. Send PayPal's cert



SKIP: skipping Messages to Java Clients

Network attacker impersonates api.paypal.com to a JSSE client

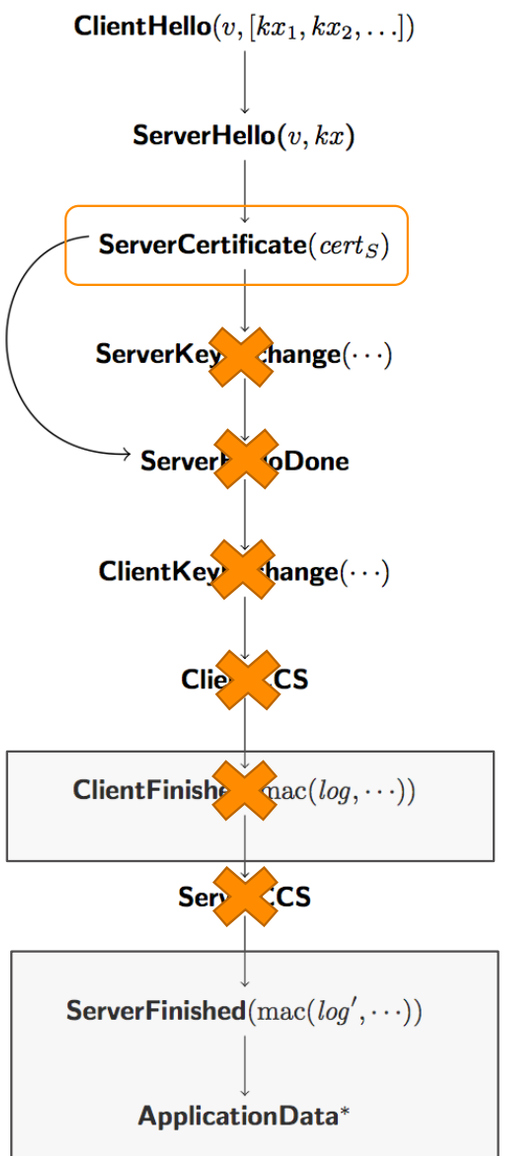
1. Send PayPal's cert
2. SKIP ServerKeyExchange (bypass server signature)
3. SKIP ServerHelloDone



SKIP: skipping Messages to Java Clients

Network attacker impersonates api.paypal.com to a JSSE client

1. Send PayPal's cert
2. SKIP ServerKeyExchange (bypass server signature)
3. SKIP ServerHelloDone
4. SKIP ServerCCS (bypass encryption)
5. Send ServerFinished using uninitialized MAC key (bypass handshake integrity)
6. Send ApplicationData (unencrypted) as S.com



State Machine Attacks

Impact of SKIP on Java TLS Clients

- A network attacker can impersonate *any* server (Paypal, Amazon, Google) to *any* Java TLS client
- Affects all versions of Java until Jan 2015 CPU

Many other State Machine bugs in TLS libraries

- FREAK: combines crypto weakness, protocol flaw, and implementation bug

Recap: What goes wrong in TLS Deployments?

Crypto Weaknesses

- RC4, 3DES, MD5, PKCS#1 v1.5

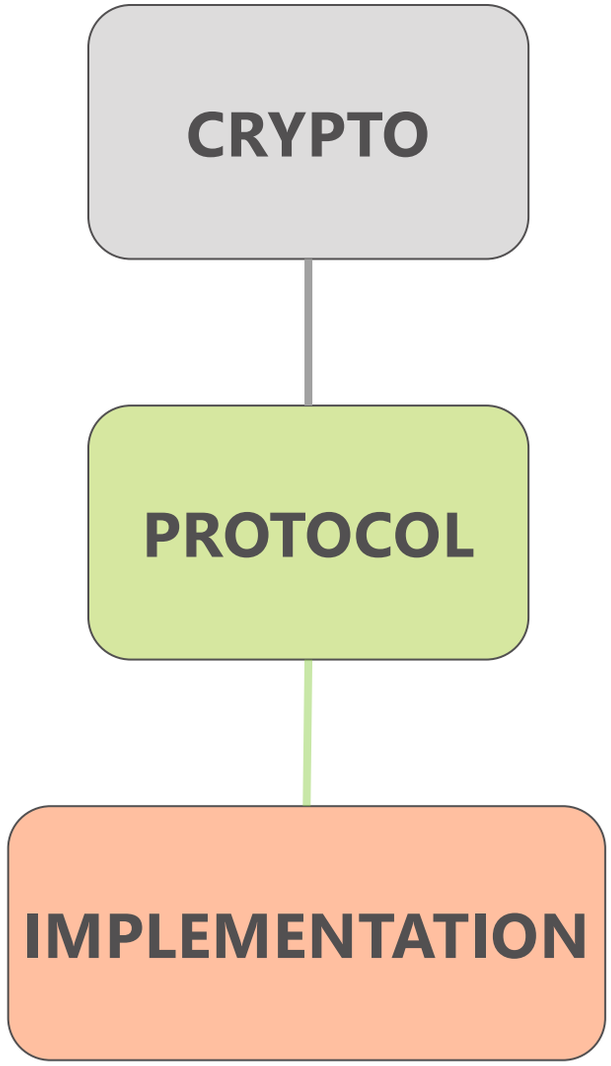
Protocol Design Flaws

- Downgrade attacks, Transcript collisions

Implementation Bugs

- State machine bugs, Heartbleed

Often, a mix of all of the above!



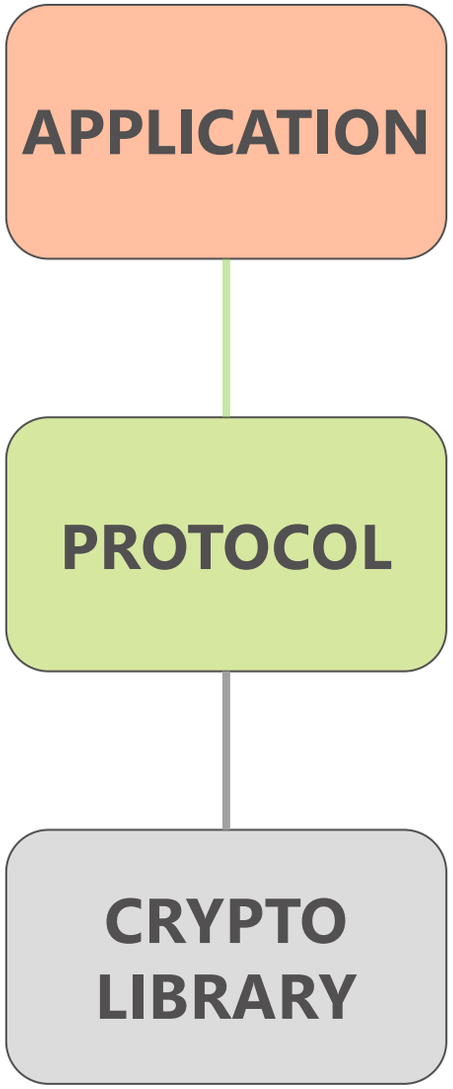
How do we fix it?

Verify protocol software

- Verified Crypto Library (HACL*)
- Verified Protocol Code (DY*)

Today:

- **Motivating Example:** HACL*
- Brief taste of verification in F*



Reading Materials

- F* language and tutorial: http://fstar-lang.org
- *Dependent Types and Multi-Monadic Effects in F**. ACM POPL 2016.
- *Verified Low-Level Programming Embedded in F**. ACM ICFP 2017.
- *HACL*: A Verified Modern Cryptographic Library*. ACM CCS 2017.
- *EverCrypt: A Fast, Verified, Cross-Platform Cryptographic Provider*. IEEE S&P 2020.
- *HACLxN: Verified Generic SIMD Crypto (for all your favourite platforms)*. ACM CCS 2020.

Verified Crypto Libraries:
HACL*

Towards High-Assurance Crypto Libraries

Crypto code is easy to get wrong and hard to test well

- memory safety bugs [CVE-2018-0739, CVE-2017-3730]
- side-channel leaks [CVE-2018-5407, CVE-2018-0737]
- arithmetic bugs [CVE-2017-3732, CVE-2017-3736]

Formal verification can systematically prevent bugs

- *Many tools*: **F***, Cryptol/Saw, VST, Fiat-Crypto, Vale, Jasmin
- But verification often requires (PhD-level) manual effort

How do we scale verification up to full crypto libraries?

- Low-level platform specific optimizations for a suite of algorithms

Writing Verified Crypto Code

CRYPTO STANDARD
(IETF/NIST)

↓

ALGORITHM
PSEUDOCODE

↓

IMPLEMENTATION
(C, 200 loc)

```c
static void chacha20_core(chacha_buf *output, const u32 input[16])
{
    u32 x[16];
    int i;
    const union {
        long one;
        char little;
    } is_endian = { 1 };

    memcpy(x, input, sizeof(x));

    for (i = 20; i > 0; i -= 2) {
        QUARTERROUND(0, 4, 8, 12);
        QUARTERROUND(1, 5, 9, 13);
        QUARTERROUND(2, 6, 10, 14);
        QUARTERROUND(3, 7, 11, 15);
        QUARTERROUND(0, 5, 10, 15);
        QUARTERROUND(1, 6, 11, 12);
        QUARTERROUND(2, 7, 8, 13);
        QUARTERROUND(3, 4, 9, 14);
```

Obviously correct?

unless we introduced
a buffer overflow,
or a timing leak

Writing Verified Crypto Code

CRYPTO STANDARD
(IETF/NIST)

↓

ALGORITHM
PSEUDOCODE

↓

IMPLEMENTATION
(C, 500 loc)

2.

```c
while (len >= POLY1305_BLOCK_SIZE) {
    /* h += m[i] */
    h0 = (u32)(d0 = (u64)h0 + U8TOU32(inp + 0));
    h1 = (u32)(d1 = (u64)h1 + (d0 >> 32) + U8TOU32(inp + 4));
    h2 = (u32)(d2 = (u64)h2 + (d1 >> 32) + U8TOU32(inp + 8));
    h3 = (u32)(d3 = (u64)h3 + (d2 >> 32) + U8TOU32(inp + 12));
    h4 += (u32)(d3 >> 32) + padbit;

    /* h *= r "%" p, where "%" stands for "partial remainder" */
    d0 = ((u64)h0 * r0) +
         ((u64)h1 * s3) +
         ((u64)h2 * s2) +
         ((u64)h3 * s1);
    d1 = ((u64)h0 * r1) +
         ((u64)h1 * r0) +
         ((u64)h2 * s3) +
         ((u64)h3 * s2) +
         (h4 * s1);
    d2 = ((u64)h0 * r2) +
         ((u64)h1 * r1) +
         ((u64)h2 * r0) +
         ((u64)h3 * s3) +
         (h4 * s2);
    d3 = ((u64)h0 * r3) +
         ((u64)h1 * r2) +
         ((u64)h2 * r1) +
         ((u64)h3 * r0) +
         (h4 * s3);
    h4 = (h4 * r0);
```

etic

Optimized 32-bit Code

a lot more code, with possible carry propagation bugs, or buffer overflows, or timing leaks.

Modular Arithmetic in Crypto

$$g^{xy} \mod p$$

$0 < x,y < p$
g fixed

Large Prime
(e.g. $2^{255} - 19$)

Implementing Modular Exponentiation

$$a^b \mod n$$

$$= a \times a \times \ldots \times a \mod n$$

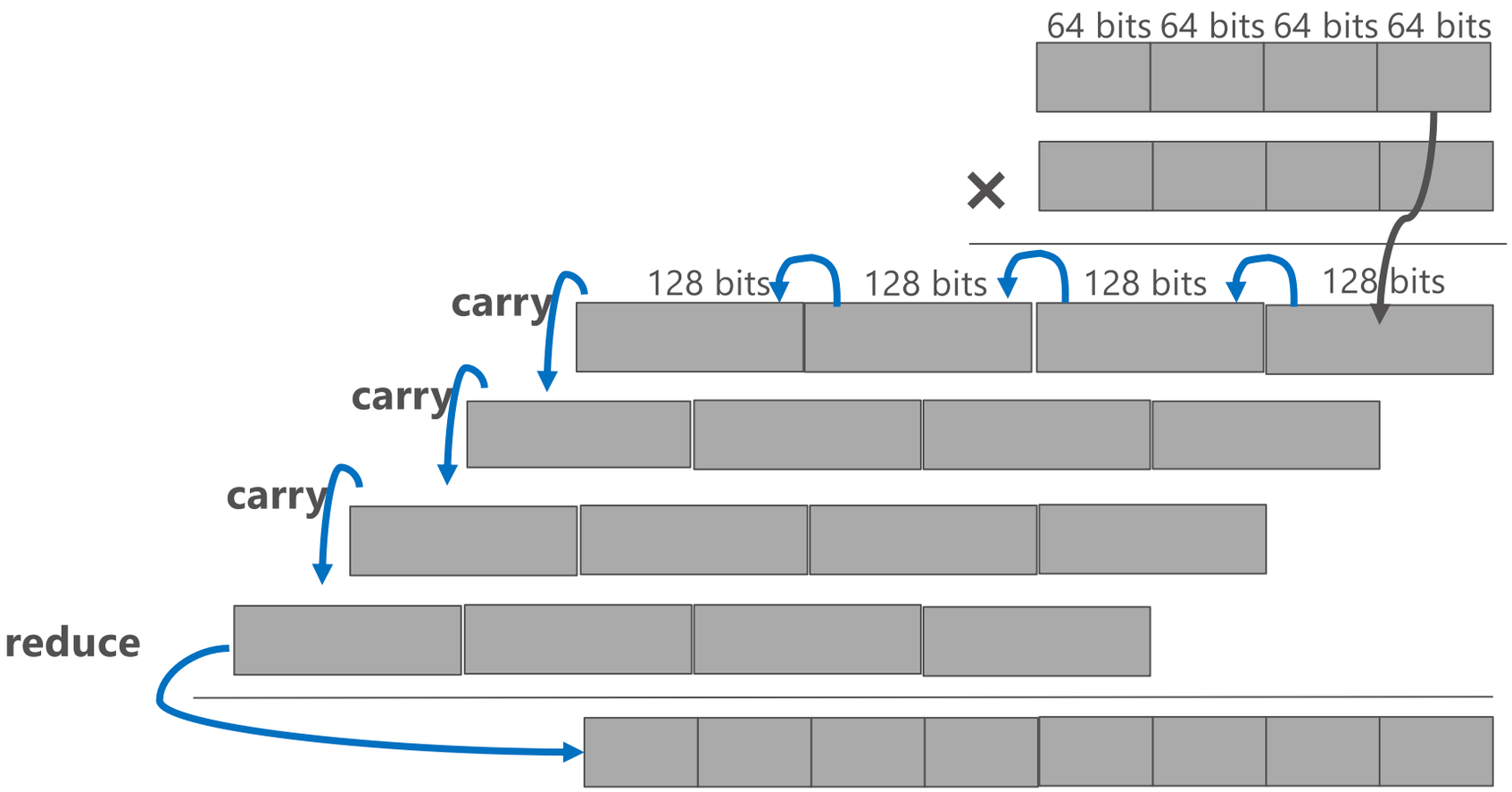
Big Integer (≥ 256-bit) arithmetic

Textbook Multiplication

```
        1101
  ×     1010
        ----
carry   0000
carry  1101
carry 0000
+    1101
  --------
  10000010
```

13
× 10
= 130

256-bit Modular Multiplication on 64-bit Computers



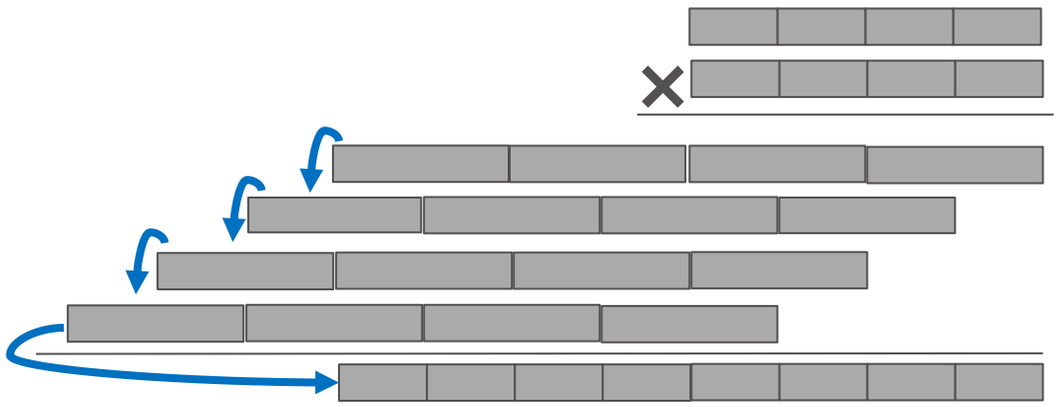
256-bit Modular Multiplication on 64-bit Computers

What can go wrong?

- Integer overflow (undefined output)
- Buffer overflow/underflow (memory error)
- Missing carry steps (wrong answer)
- Side-channel Attack (leaks secrets)

How expensive is it?

- *Dominates crypto overhead*
- n^2 64x64 multiplications
- Long intermediate arrays
- Many carry steps



Optimizations vs. Side-Channel Leaks

```
      1101
  ×   1010
      ----
     1101
+ 1101
--------
10000010
```

Skipping steps is faster

- Fewer additions, carries

… but may leak information

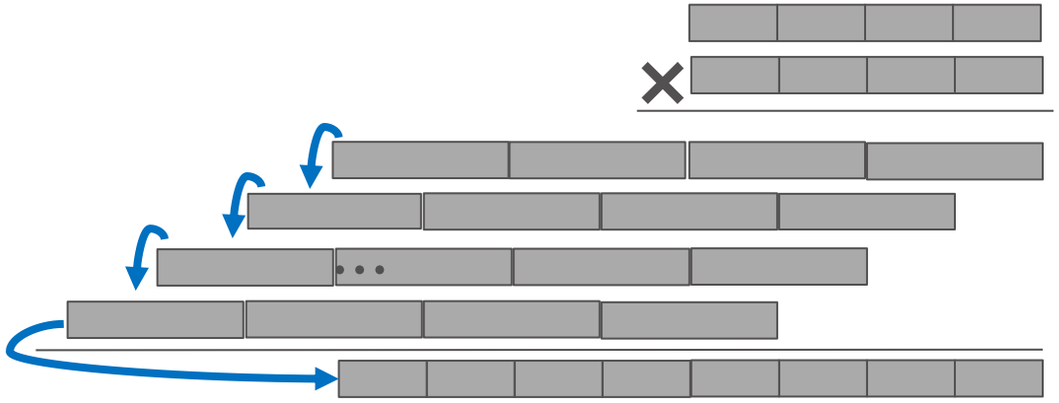
- Runtime proportional to number of 1s in 1010
- Attacker can observe runtime to guess input
- Input may be secret key!

Optimizations in Modular Arithmetic

Many prime-specific optimizations

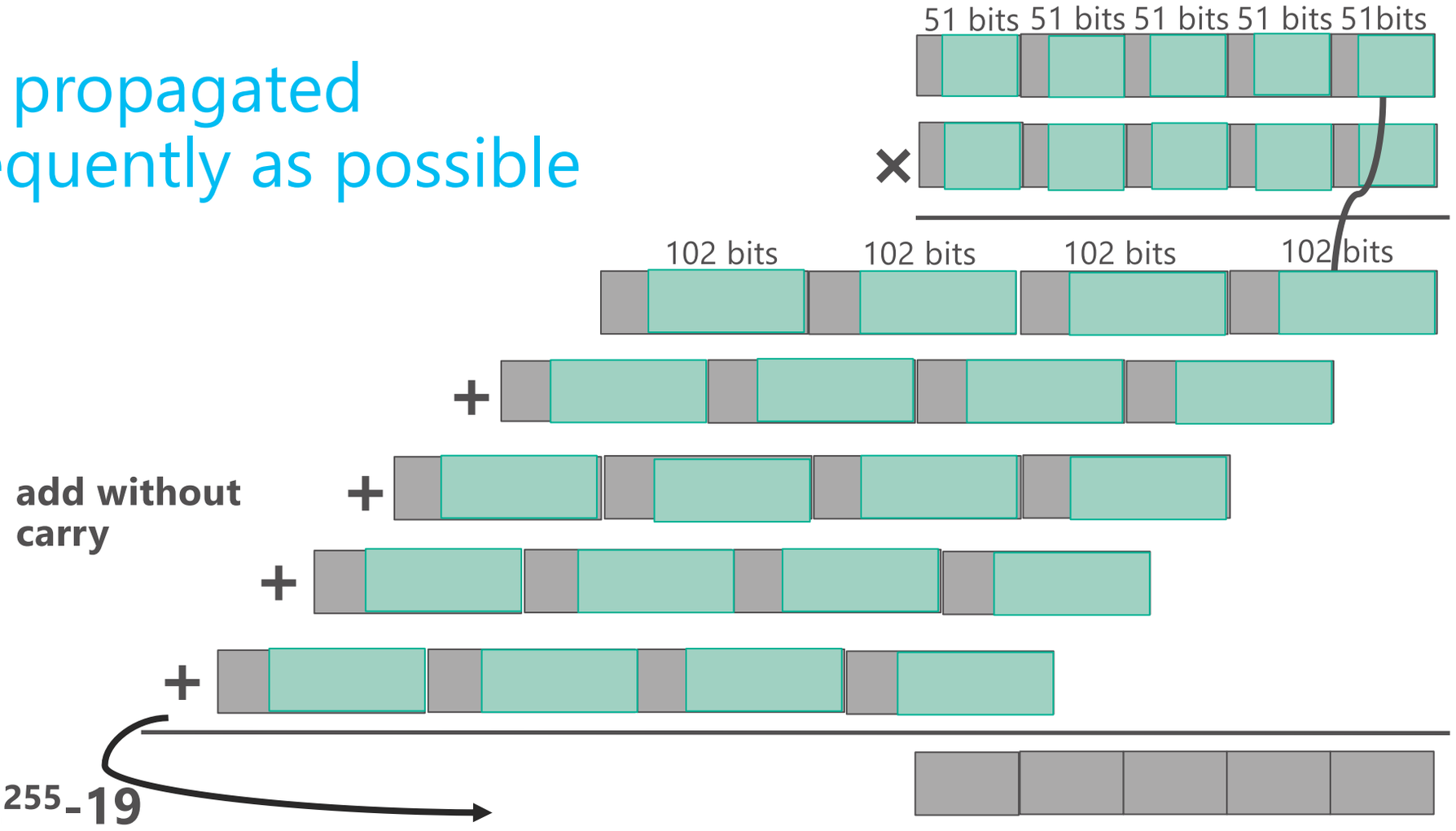
- Trade-off multiplication vs modular reduction
- Use only 51 out of 64 bits to avoid carries
- Precompute reusable intermediate values
- Parallelize (vectorize) multiplication and squaring

Complex optimizations increase chances of bugs!



Unsaturated Arithmetic for Curve25519

Carries propagated
as infrequently as possible



Many bugs in optimized bignum code

[2013] Bug in amd-64-64-24k Curve25519

> *"Partial audits have revealed a bug in this software (r1 += 0 + carry should be r2 += 0 + carry in amd64-64-24k) that would not be caught by random tests; this illustrates the importance of audits."*
>
> – D.J. Bernstein, W. Janssen, T. Lange, and P. Schwabe (2013)

[2014] Arithmetic bug in TweetNaCl's Curve25519

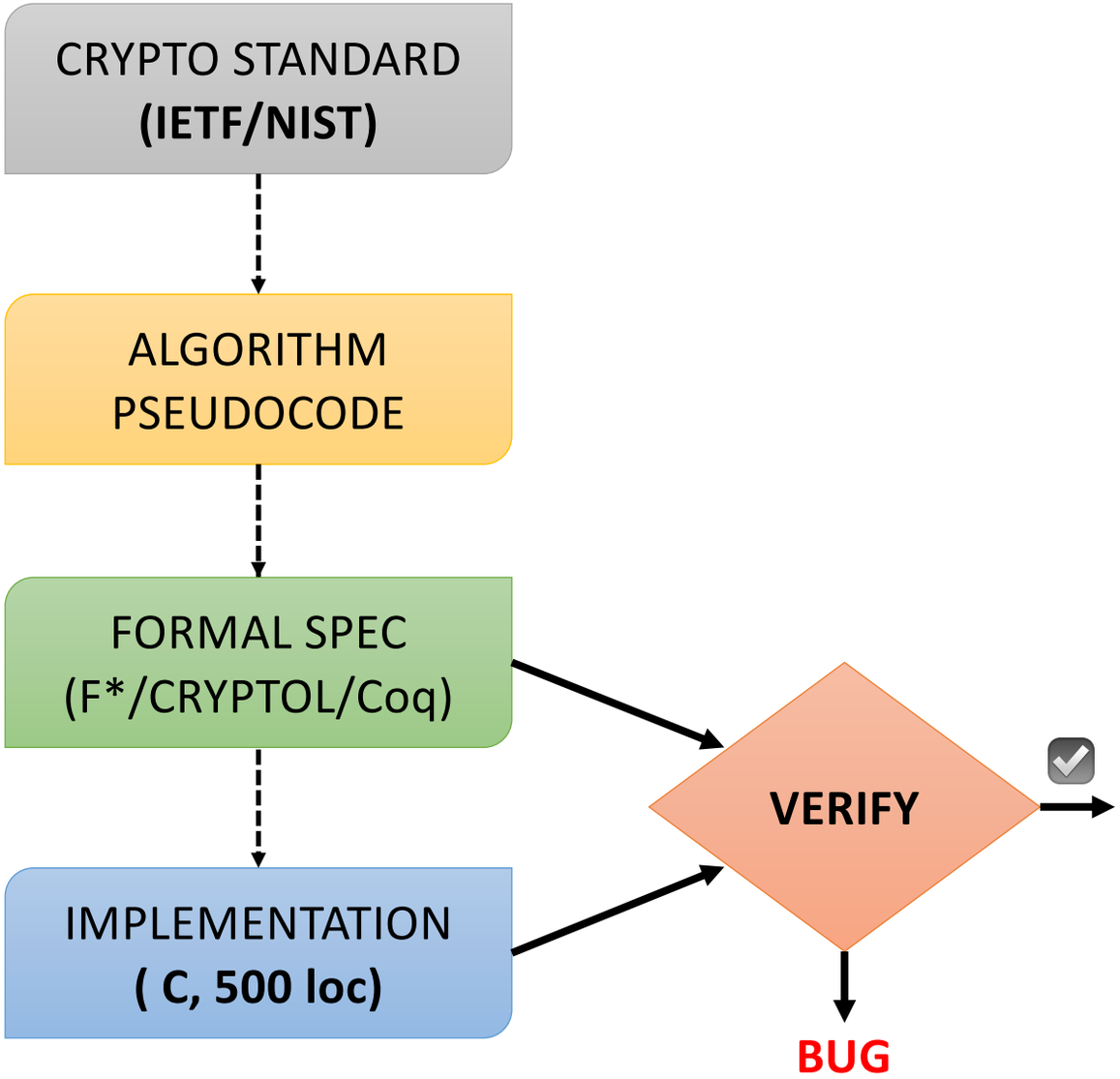
[2014] Carry bug in Langley's Donna-32 Curve25519

[2016] Arithmetic bug in OpenSSL Poly1305

[2017] Arithmetic bug in Mozilla NSS GF128

+ Many memory bugs, side-channel leaks, …

Writing Verified Crypto Code



Verification Guarantees

1. *Functional Correctness*
2. *Memory Safety*
3. *Secret Independence (constant-time)*

HACL*: a verified C crypto library

[Zinzindohoé et al. ACM CCS 2017]

A growing library of verified crypto algorithms

- Curve25519, Ed25519, Chacha20, Poly1305, SHA-2, HMAC, …

Implemented and verified in F* and compiled to C

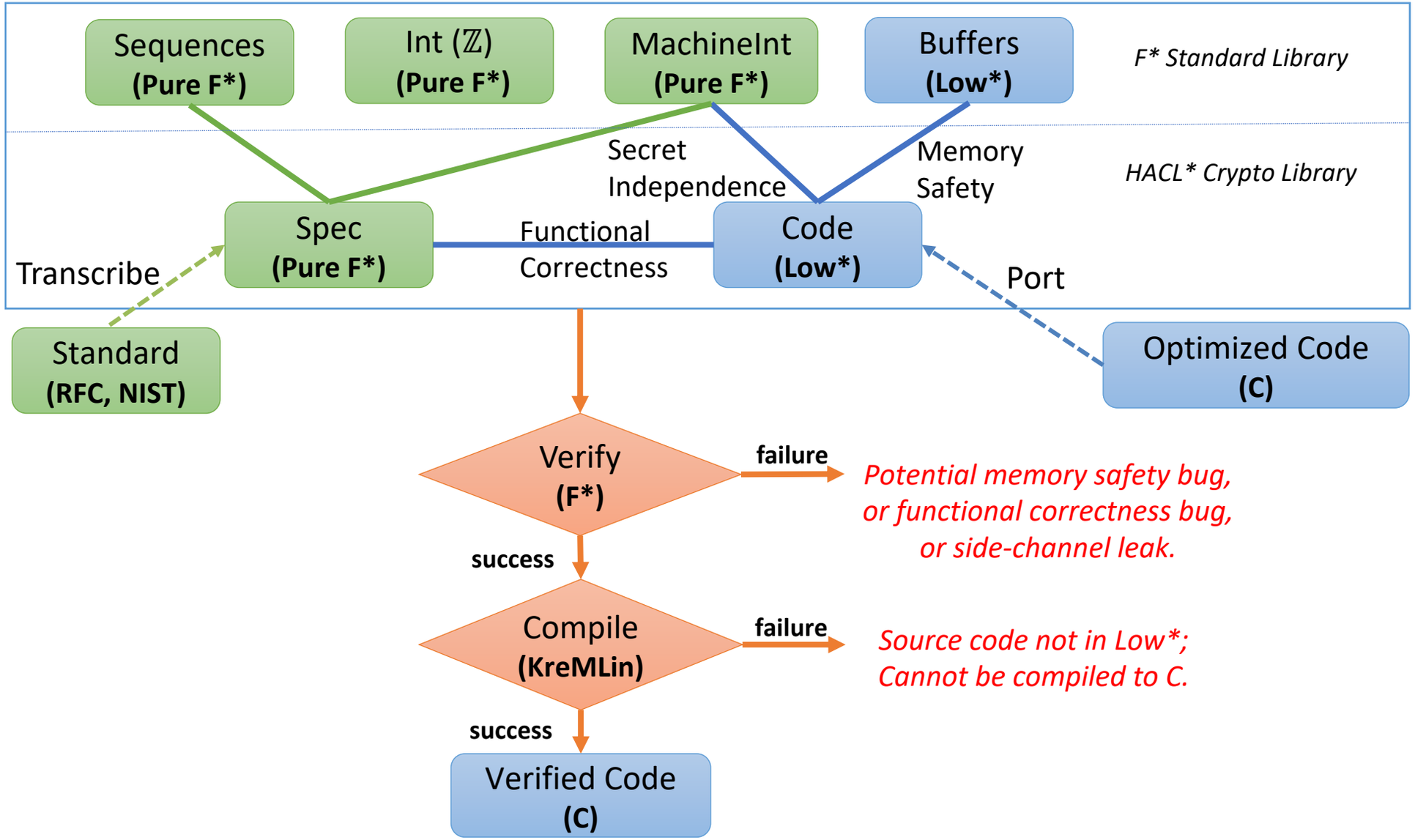
- **Memory safety** proved in the C memory model
- **Secret independence** ("constant-time") enforced by typing
- **Functional correctness** against a mathematical spec written in F*

Generates readable, portable, standalone C code

- Performance comparable to hand-written C crypto libraries
- Used in **Mozilla Firefox**, **WireGuard VPN**, **Tezos Blockchain**, …

https://github.com/project-everest/hacl-star

HACL* Verification workflow



Example: Poly1305 MAC Algorithm

Poly1305 is a message authentication code

$$poly(k, m, w_1 \ldots w_n) = m + w_1 k^1 + \ldots + w_n k^n \quad \% \ (2^{130} - 5)$$

It authenticates a data stream $w_1 \ldots w_n$ by

- Encoding it as a polynomial in the prime-field modulo $2^{130} - 5$
- Evaluating it at a point k (first part of the key)
- Masking the result with m (second part of the key)

Specifying Poly1305 in Pure F*

- Short, easy to review
- Uses arbitrary precision natural numbers

- Compiles to OCaml
- Passes RFC test-vectors

Spec.Poly1305.fst

```
module Spec.Poly1305

let prime = pow2 130 - 5
type elem = e:Z{e ≥ 0 ∧ e < prime}
let fadd (e1:elem) (e2:elem) = (e1 + e2) % prime
let fmul (e1:elem) (e2:elem) = (e1 × e2) % prime

(* Specification code *)
let encode (w:word) =
  (pow2 (8 × length w)) `fadd` (little_endian w)

let rec poly (txt:text) (r:e:elem) : Tot elem (decreases (length txt)) =
  if length txt = 0 then zero
  else
    let a = poly (Seq.tail txt) r in
    let n = encode (Seq.head txt) in
    (n `fadd` a) `fmul` r
```

-:**- Spec.Poly1305.fst All (2,0) Git-master (F✪ company)
Auto-saving...done

Efficiently Implementing Poly1305

1. Encode field elements with a 44-44-42 unsaturated representation
2. Factor out generic bignum operations (+, *) into a shared library
3. Optimized prime-specific field arithmetic in Poly1305
4. Expose an Init-Update-Finish API for incremental use

Low* code+proofs: 1508 lines (generic bignum) + 3208 lines (poly1305)

Compiled C code: 451 lines

Verifying Memory Safety by Typing

```
val fsum:
  a:felem →
  b:felem →
  Stack unit
    (requires (λ h → live h a ∧ live h b
                     ∧ disjoint a b
                     ∧ no_overflow h a b len))
    (ensures (λ h0 _ h1 → live h1 a ∧ live h1 b
                          ∧ modifies_1 a h0 h1
                          ∧ eval h1 a = eval h0 a + eval h0 b))
```

1. Ensure all pointers are live (initialized and not yet freed)
2. Verify all array accesses (access within bounds)
3. Enforce disjointness (needed for correctness)
4. Track modifications (needed for composability)

Verifying Functional Correctness

```
val fsum:
  a:felem →
  b:felem →
  Stack unit
    (requires (λ h → live h a ∧ live h b
                     ∧ disjoint a b
                     ∧ no_overflow h a b len))
    (ensures (λ h0 _ h1 → live h1 a ∧ live h1 b
                          ∧ modifies_1 a h0 h1
                          ∧ eval h1 a = eval h0 a + eval h0 b))
```

Prove that stateful code matches pure F* specification

- Relies on mathematical theory of modular arithmetic
- Simple arithmetic goals automatically verified by the SMT solver (Z3)
- Complex prime-specific optimizations proved using F*

Verifying Secret Independence

Type-based "constant-time" coding discipline

- Code cannot branch on secrets
- Code cannot use secret indices to lookup arrays
- Essentially, a crude static information flow checker via types

Prevents timing attacks within C semantics

- No guarantees on compiled assembly
- Does not guarantee absence of other side channels
- For better guarantees, see:
  Verifying Constant-Time Implementations, Almeida et al. Usenix'16

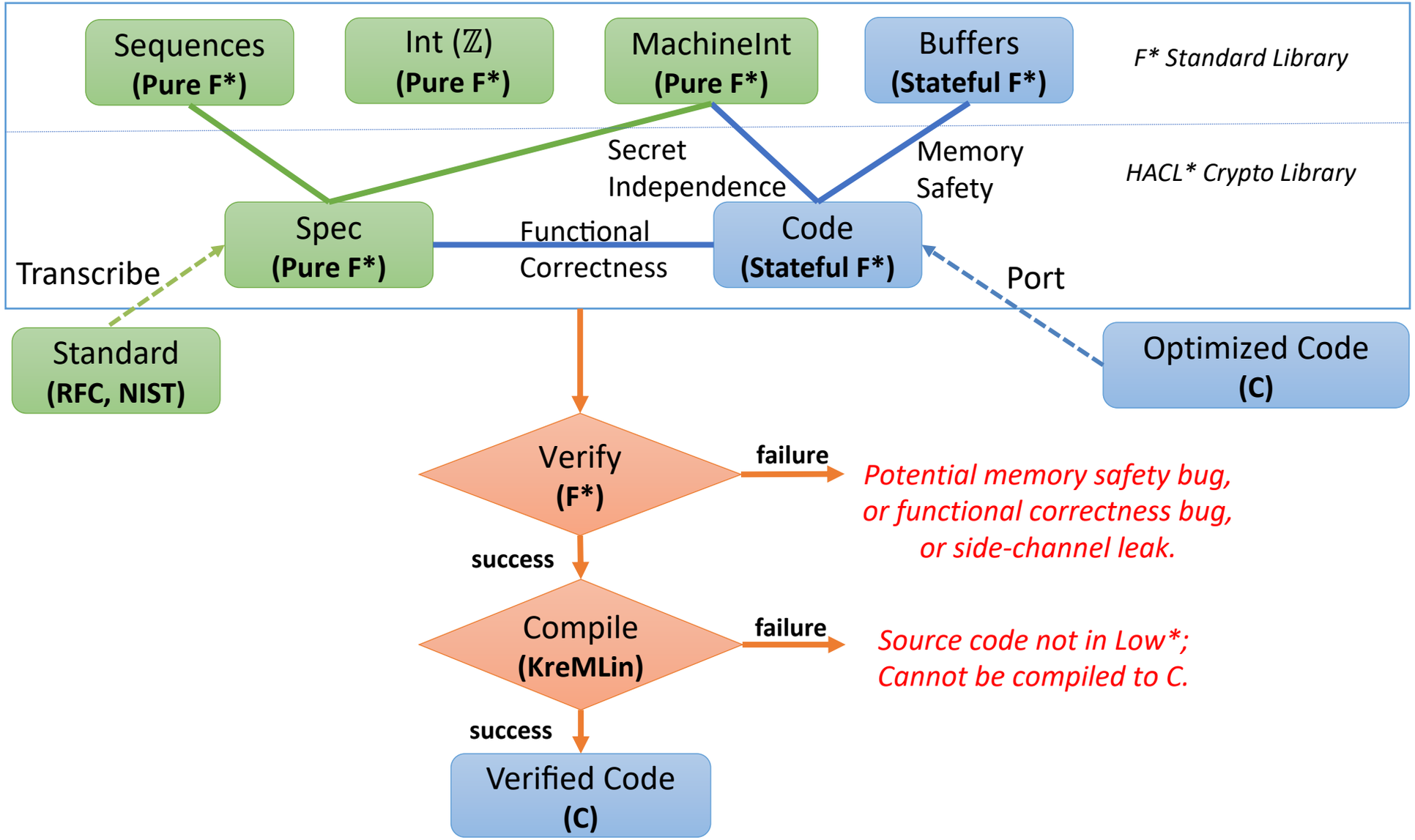
Verifying Secret Independence

```
abstract type hint64_t
val v: hint64_t → GTot uint64_t

val (+): hint64_t → hint64_t → Tot hint64_t
val (^): hint64_t → hint64_t → Tot hint64_t
// val (/): hint64_t -> hint64_t -> Tot hint64_t

type key = b:buffer uint64_t{length b = 32}
```

- Abstract types for opaque “hidden integers”
  their concrete values are only available in specifications
- Allowed constant-time operations: +, -, *, ^, &, |
  but no comparisons or divisions, or use as array indexes

  (Forbidden operations depend on target platform)

HACL* Verification Workflow



F* Code for Poly1305

Compiled C Code

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)

memory safety

math spec

Code

proof

Poly1305_64.c

File Edit Options Buffers Tools C Help

```
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 )0x3fffffffffff);
  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 )0x3fffffffffff & mask);
  acc[0] = a0_0;
  acc[1] = a1_0;
  acc[2] = a2_0;
}
```

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

Low*: High-level verification for low-level code

[Protzenko et al. ICFP 2017]

Low* is a subset of F* that mimics C programs

- Relies on a C-like memory model;
- Mostly first-order code with combinators to get C loops;
- A few low-level libraries for arrays, structs, and C base types.

When writing proofs and specifications, the programmer

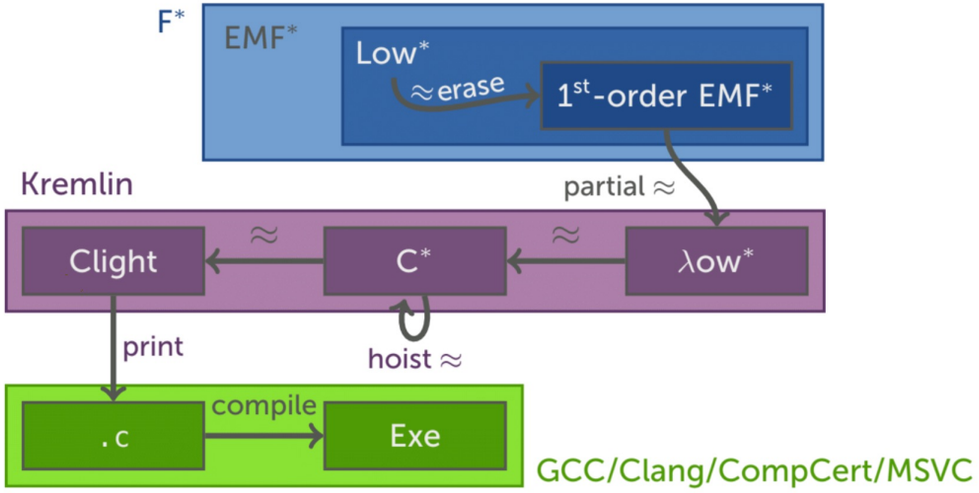
- uses all of F*, including higher-order functions and polymorphism;
- proves memory safety, correctness, cryptographic security, by adding lemmas, type annotations, auxiliary functions, etc.;
- compiles the code to a first-order program by erasing all proofs.

Motto: the code is low-level but the verification is not

KreMLin: Compiling Low* to Readable C

https://github.com/FStarLang/kremlin

- Implements a formal translation from Low* to Clight
- Hand-written proof of trace preservation
  - Compilation preserves side-channel guarantees [Protzenko et al. ICFP 2017]
- Lots of engineering to generate readable, efficient C code



Low* Code for Poly1305

Compiled C Code

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'
```

memory safety

math spec

Code

proof

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

Poly1305_64.c

File Edit Options Buffers Tools C Help

```
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 )0x3fffffffffff);
  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 )0x3fffffffffff & mask);
  acc[0] = a0_0;
  acc[1] = a1_0;
  acc[2] = a2_0;
}
```

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

HACL* Verification

- Share verified libraries across various primitives
- Verified optimizations SIMD vectorization, prime-specific arithmetic
- Significant manual effort in initial release, now significantly reduced.

Algorithm	Spec (F* loc)	Code+Proofs (Low* loc)	C Code (C loc)	Verification (s)
Salsa20	70	651	372	280
Chacha20	70	691	243	336
Chacha20-Vec	100	1656	355	614
SHA-256	96	622	313	798
SHA-512	120	737	357	1565
HMAC	38	215	28	512
Bignum-lib	-	1508	-	264
Poly1305	45	3208	451	915
X25519-lib	-	3849	-	768
Curve25519	73	1901	798	246
Ed25519	148	7219	2479	2118
AEAD	41	309	100	606
SecretBox	-	171	132	62
Box	-	188	270	43
Total	801	22,926	7,225	9127

Table 1: HACL* code size and verification times

HACL* Performance

Algorithm	HACL*	OpenSSL	libsodium	TweetNaCl	OpenSSL (asm)
SHA-256	13.43	16.11	12.00	-	7.77
SHA-512	8.09	10.34	8.06	12.46	5.28
Salsa20	6.26	-	8.41	15.28	-
ChaCha20	6.37 (ref) 2.87 (vec)	7.84	6.96	-	1.24
Poly1305	2.19	2.16	2.48	32.65	0.67
Curve25519	154,580	358,764	162,184	2,108,716	-
Ed25519 sign	63.80	-	24.88	286.25	-
Ed25519 verify	57.42	-	32.27	536.27	-
AEAD	8.56 (ref) 5.05 (vec)	8.55	9.60	-	2.00
SecretBox	8.23	-	11.03	47.75	-
Box	21.24	-	21.04	148.79	-

- 20% faster than previous code in Firefox
- As fast as hand-optimized C code in OpenSSL
- 0%-30% slower than equivalent hand-tuned assembly

Can we make it faster?

Verified Assembly,
Vectorized Algorithms

Can we make it easier?

Smaller Specs and Code,
Fewer Proof Annotations

HACL*: estimating verification effort

CHACHA20

High-level F* Spec	70 lines
Verified F* Code	691 lines
Generated C Code	285 lines
Proof Annotations	**406 lines**

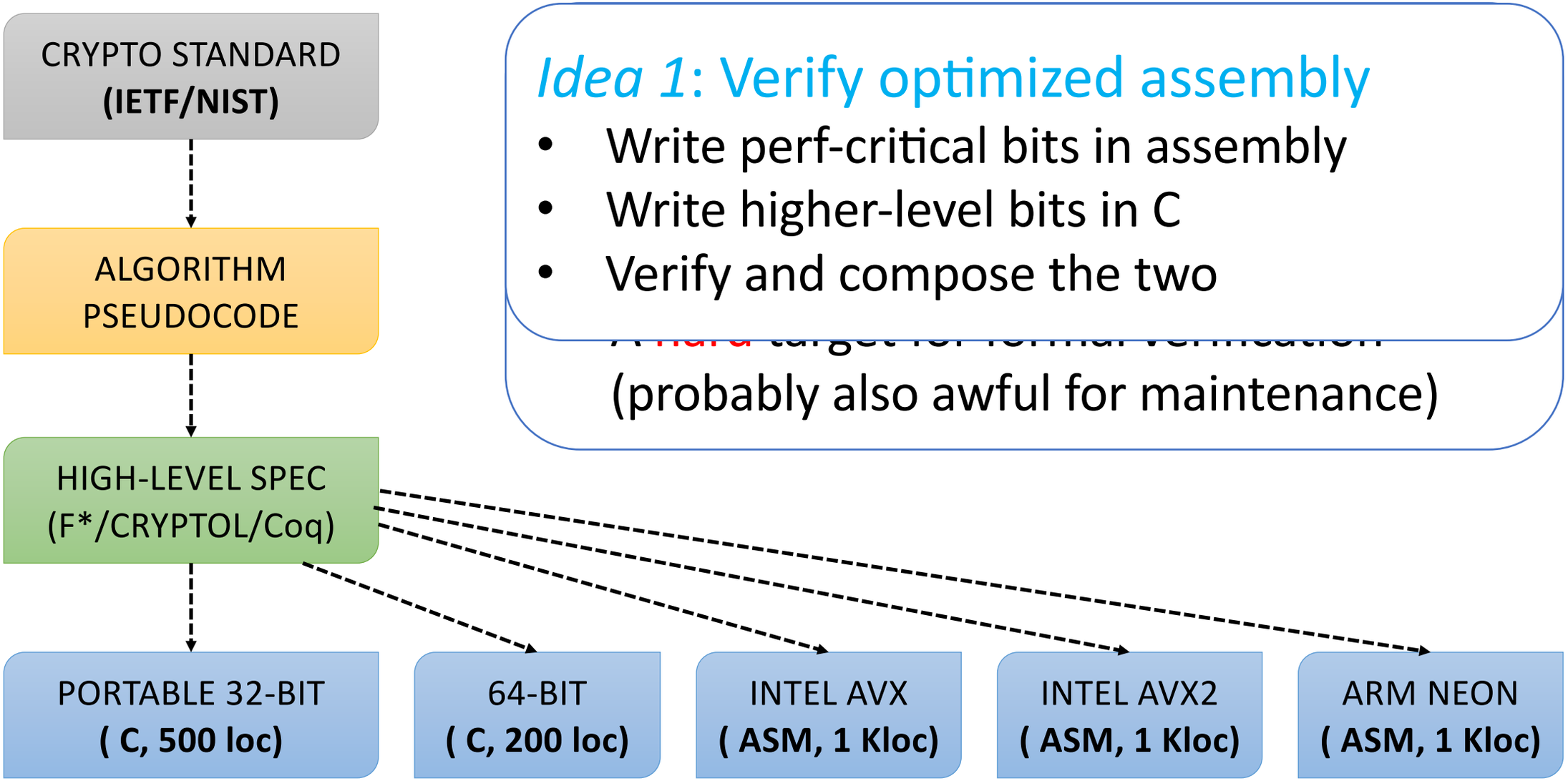
POLY1305

High-level F* Spec	45 lines
Verified F* Code	3967 lines
Generated C Code	451 lines
Proof Annotations	**3516 lines**

Every line of verified C requires 2x-7x lines of proof

Complex mathematical reasoning interleaved with many boring steps

Platform-Specific Optimizations



Idea 1: Verify optimized assembly

- Write perf-critical bits in assembly
- Write higher-level bits in C
- Verify and compose the two

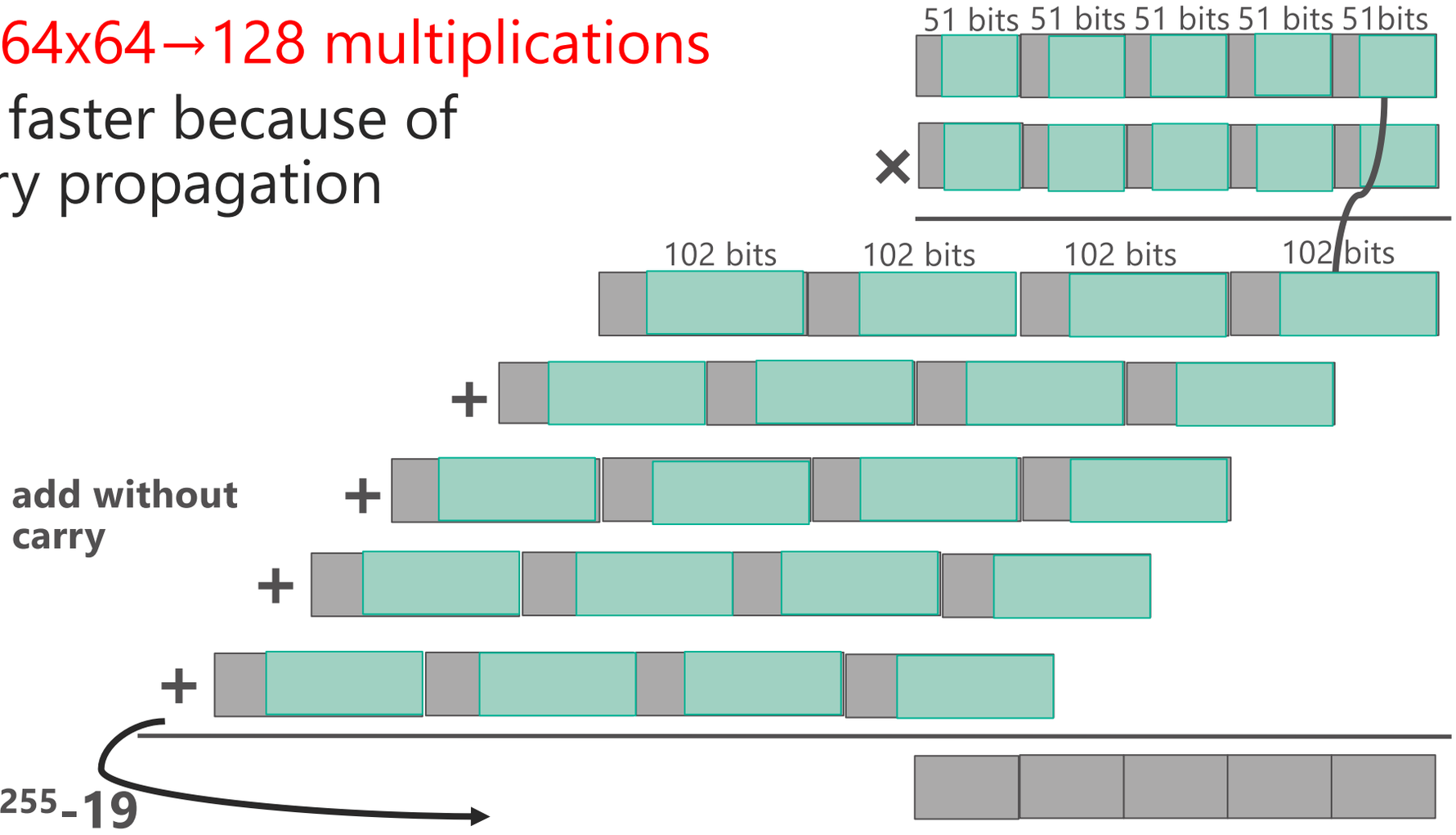
A hard target for formal verification

(probably also awful for maintenance)

Recall: Unsaturated Arithmetic for Curve25519

9 more 64x64→128 multiplications

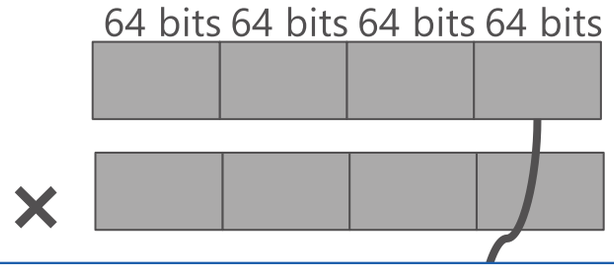
but still faster because of less carry propagation



Saturated 64-bit Arithmetic with Intel ADX

[Oliveira et al, SAC 2017]

An instruction set with 2 carry flags can significantly reduce the cost of carry propagation!



Intel(R) Xeon(R) CPU E3-1505M v5 @ 2.80GHz
donna64: 160942 cycles per call
hacl64: 140902 cycles per call
fiat64: 144106 cycles per call
sandy2x: 136074 cycles per call
precomp_bmi2: 121350 cycles per call
precomp_adx: 117676 cycles per call
amd64: 143628 cycles per call
fiat32: 307971 cycles per call
donna32: 544254 cycles per call

New speed record?

Measurements using Jason Donenfeld's Linux Kernel Benchmarking Suite for WireGuard.

Saturated 64-bit Arithmetic with Intel ADX

Armando Faz Hernández about a year ago

> *Post by Jason A. Donenfeld*
> Hi Armando,
> I've started importing your precomputation implementation into kernel space for use in kbench9000 (and in WireGuard and the kernel crypto library too, of course).
> - The first problem remains the license. The kernel requires GPLv2-compatible code. GPLv3 isn't compatible with GPLv2. This isn't up to me at all, unfortunately, so this stuff will have to be licensed differently in order to be useful.

The rfc7748_precomputed library is now released under LGPLv2.1.
We are happy to see our code integrated in more projects.

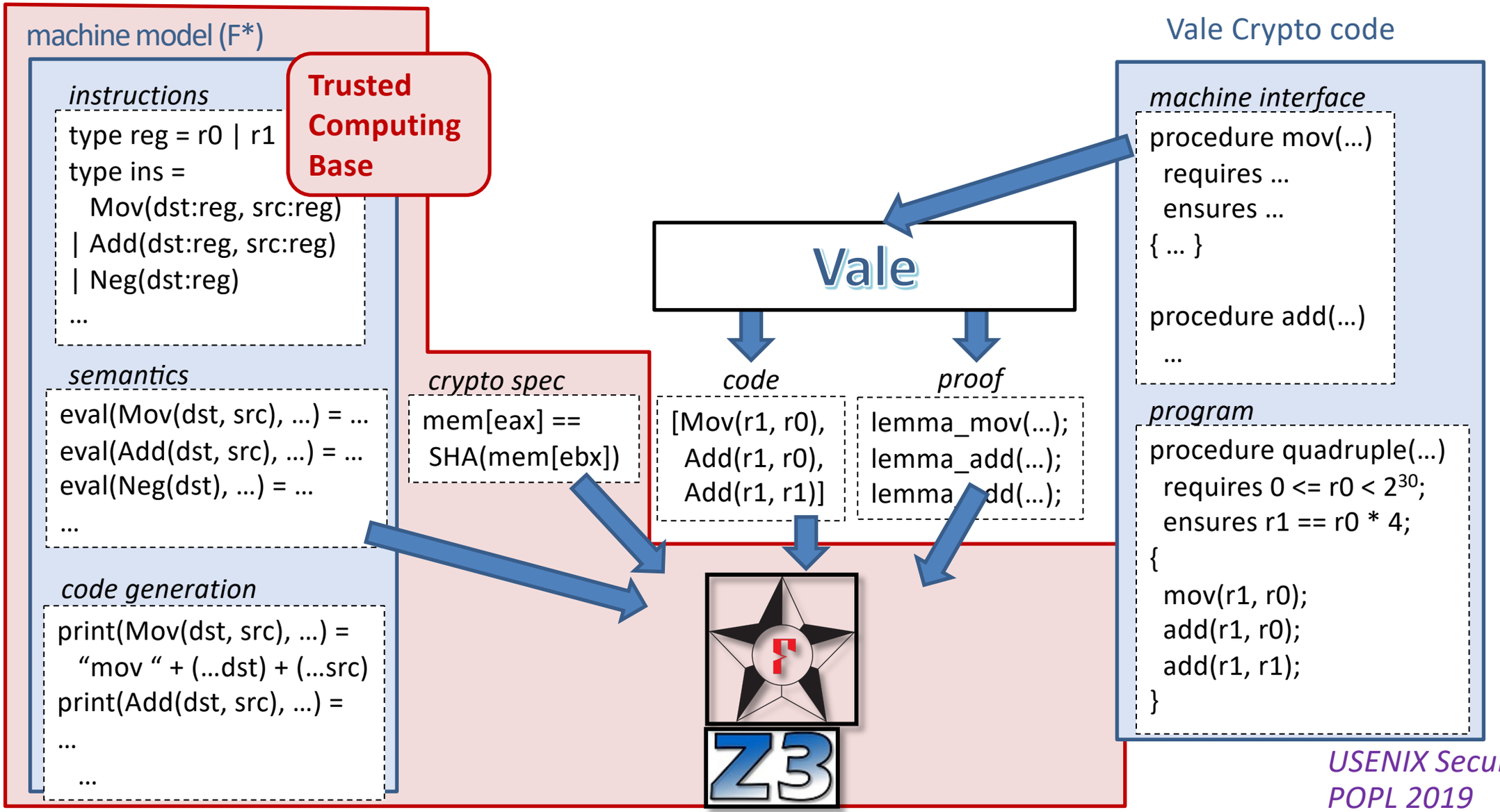
> *Post by Jason A. Donenfeld*
> - It looks like the precomputation implementation is failing some unit tests! Perhaps it's not properly reducing incoming public points? There's the vector if you'd like to play with it. The other test vectors I have do pass, though, which is good I suppose.

Thanks, for this observation. The code was missing to handle some carry bits, producing incorrect outputs for numbers between 2p and 2^256. Now, I have rewritten some operations for GF(2^255-19) considering all of these cases. More tests were added and fuzz test against HACL implementation.

Still hard to get right!

Vale: extensible, assembly language verification

[Usenix 2017, POPL 2019]



USENIX Security 2017, POPL 2019

Verified Assembly for 256-bit Multiplication

Implement core arithmetic in Vale x86

- Use ADX+BMI2 instructions
- Verify correctness + constant-time

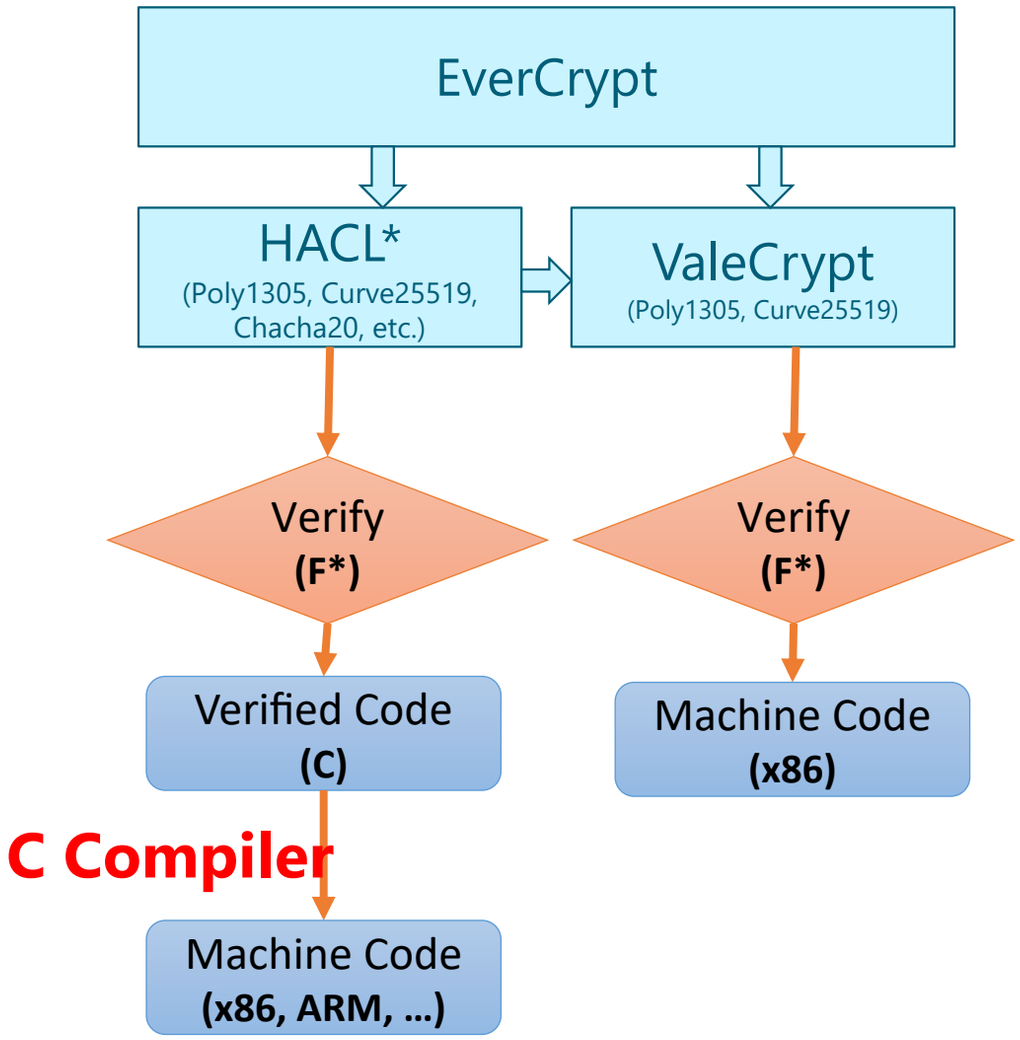
Implement curve operations in HACL*

- Use standard C coding style
- Optimize add/double formulas, montgomery ladder, etc.

Prove that composition of Vale and F* code meets Curve25519 spec

```
mul2:
  push %r12
  push %r13
  push %r14
  mov %rdx, %rcx
  movq 0(%rsi), %rdx
  mulxq 0(%rcx), %r8, %r9
  xor %r10, %r10
  movq %r8, 0(%rdi)
  mulxq 8(%rcx), %r10, %r11
  adox %r9, %r10
  movq %r10, 8(%rdi)
  mulxq 16(%rcx), %r12, %r13
  adox %r11, %r12
  mulxq 24(%rcx), %r14, %rdx
  adox %r13, %r14
  mov $0, %rax
  adox %rdx, %rax
  movq 8(%rsi), %rdx
```

EverCrypt: a crypto provider with Vale + HACL*



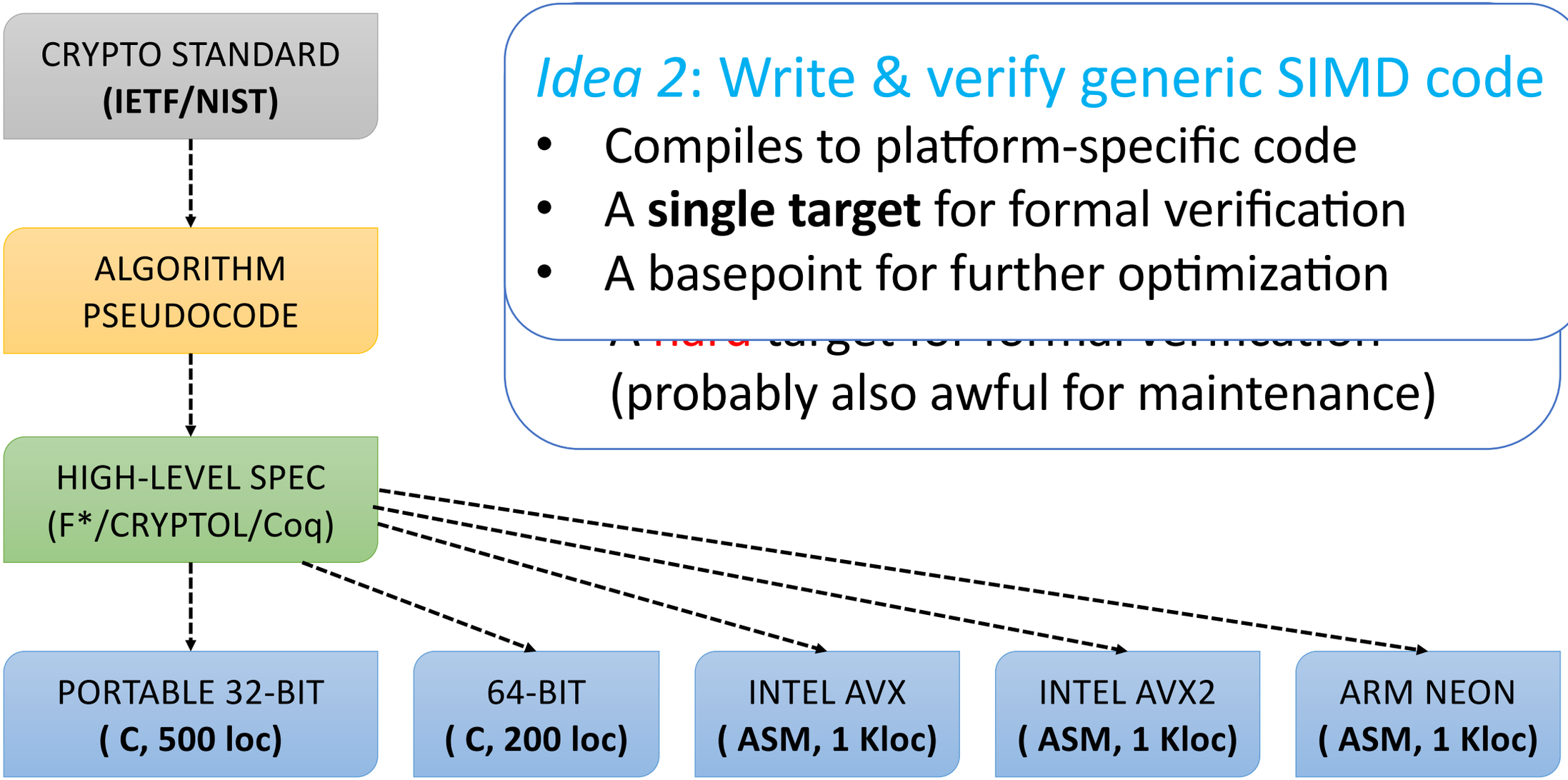
Algorithm	C version	Targeted ASM version
AEAD		
AES-GCM		AES-NI + PCLMULQDQ + AVX
ChachaPoly	yes	
Hashes		
MD5	yes	
SHA1	yes	
SHA2	yes	SHA-EXT (for SHA2-224+SHA2-256)
MACS		
HMAC	yes	agile over hash
Poly1305	yes	X64
Key Derivation		
HKDF	yes	agile over hash
ECC		
Curve25519	yes	BMI2 + ADX
Ciphers		
Chacha20	yes	
AES128, 256		AES NI + AVX
AES-CTR		AES NI + AVX

Curve25519 Performance: Vale + HACL*

Implementation	Radix	Language	CPU cy.
donna64	51	64-bit C	159634
fiat-crypto [24]	51	64-bit C	145248
amd64-64	51	Intel x86_64 asm	143302
sandy2x	25.5	Intel AVX asm	135660
HACL* portable	51	64-bit C	135636
openssl*	64	Intel ADX asm	118604
Oliveira et al. [45]	64	Intel ADX asm	115122
EverCrypt: Vale + HACL*	64	64-bit C + Intel ADX asm	113614

Figure 11. Performance comparison between Curve25519 Implementations.

Other Platform-Specific Optimizations



Idea 2: Write & verify generic SIMD code

- Compiles to platform-specific code
- A **single target** for formal verification
- A basepoint for further optimization

(probably also awful for maintenance)

HACL* Vectorization Performance

CHACHA20

32-bit Scalar	4 cy/b
128-bit Vectorized (AVX)	1.5 cy/b
256-bit Vectorized (AVX2)	**0.79 cy/b**
Fastest Assembly (OpenSSL AVX2)	**0.75 cy/b**

POLY1305

32-bit Scalar	1.5 cy/b
128-bit Vectorized (AVX)	0.75 cy/b
256-bit Vectorized (AVX2)	**0.39 cy/b**
Fastest Assembly (OpenSSL AVX2)	**0.34 cy/b**

Measurements with gcc-7 on Intel i7-7560 (Skylake) running Ubuntu 18.10

Estimating Verification Effort

CHACHA20

hacspec	150 lines
Vectorized algorithm	500 lines
Correctness proofs	700 lines
Vectorized code	500 lines
Total Proof Effort	1700 lines
Generated C code	3700 lines

POLY1305

hacspec	80 lines
Vectorized algorithm	450 lines
Correctness proofs	2000 lines
Vectorized code	1500 lines
Total Proof Effort	4000 lines
Generated C code	16000 lines

Effort roughly the same as verifying one implementation

HACL* Code in Deployment

Conclusions

End-to-End verification of protocol stacks is now feasible

- Using proof-oriented programming languages like F*
- Crypto (HACL*), Parsers (EverParse), Protocols (miTLS, DY*)

Area is still maturing and is under active research

- Verifying optimized low-level code in C and assembly
- Verifying protocols that use ZK proofs, or MPC, or FHE, or PQ

Many open problems for future work

- Proving the absence of side channel attacks
- Verifying code written by non-verification experts