Verified Crypto for Verified Protocols

Karthik Bhargavan
http://prosecco.inria.fr

+ many many others
(INRIA, Microsoft Research, …)

Transport Layer Security (1994—)

Widely-used secure channel protocol

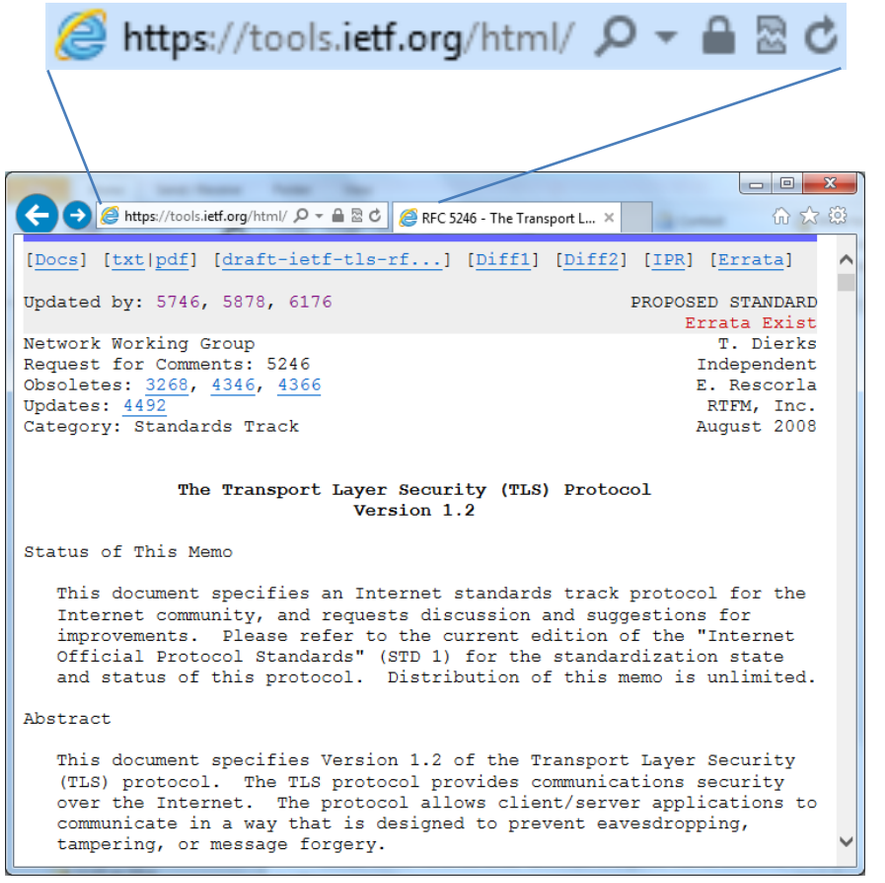
HTTPS, 802.1x, VPNs, files, mail, VoIP, …

20 years of attacks and fixes

1994 Netscape's Secure Sockets Layer
1996 SSLv3
1999 TLS1.0 (RFC2246)
2006 TLS1.1 (RFC4346)
2008 TLS1.2 (RFC5246)
2018 TLS1.3 (RFC 8846)

Many implementations

OpenSSL, SecureTransport, NSS,
SChannel, BoringSSL, JSSE, mbedTLS, …
many bugs, attacks, patches every year



Still, many, many recent attacks

• 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	[Oct'16]

Root causes of recent attacks

Weak (obsolete) crypto algorithms

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

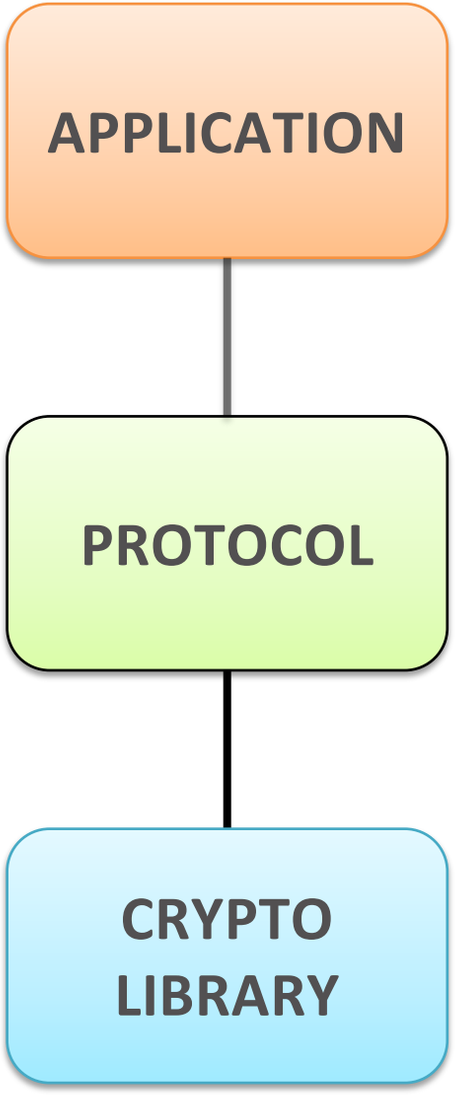
Protocol flaws in little-used corner cases

- E.g. Downgrades, Transcript collisions

Implementation bugs in protocol or crypto code

- E.g. Memory safety, Side channels, State machine bugs

Often, a mix of all of the above!



Formal Verification Can Help

Tuesday

Verified crypto protocol designs

- Symbolic analysis to find attacks [ProVerif, Tamarin]
- Cryptographic proofs of security [CryptoVerif, F*, EasyCrypt]

Verified crypto protocol software

Today

- Verified crypto libraries [F*, Coq, VST, Cryptol/SAW]
- Verified protocol implementations [F*, FCF/Coq]

Protocol Implementation Bugs:
State Machine Attacks

Many, many modes of TLS

Protocol versions

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

Key exchanges

- ECDHE, FFDHE, RSA, PSK, …

Authentication modes

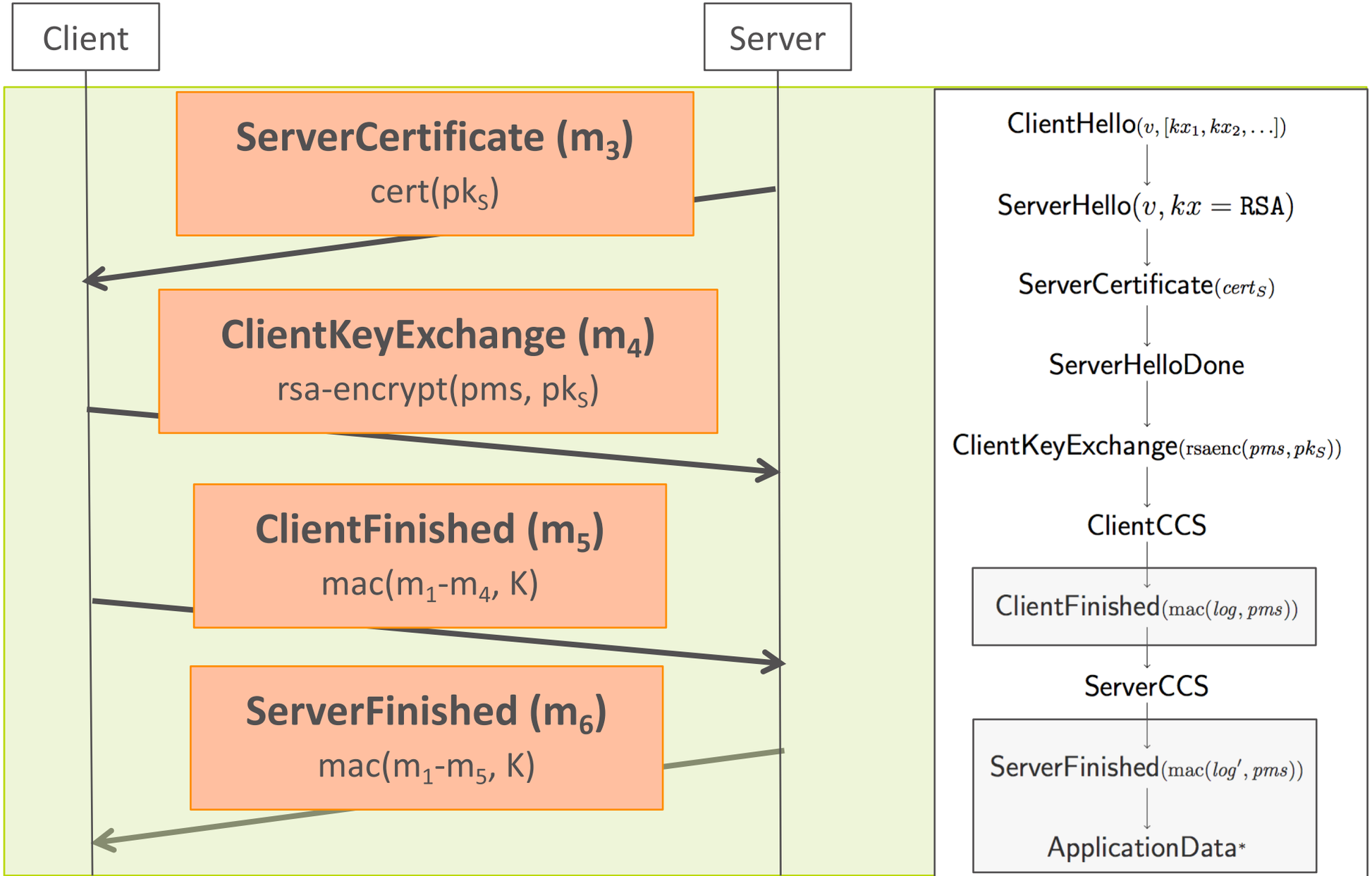
- ECDSA, RSA signatures, PSK,…

Authenticated Encryption Schemes

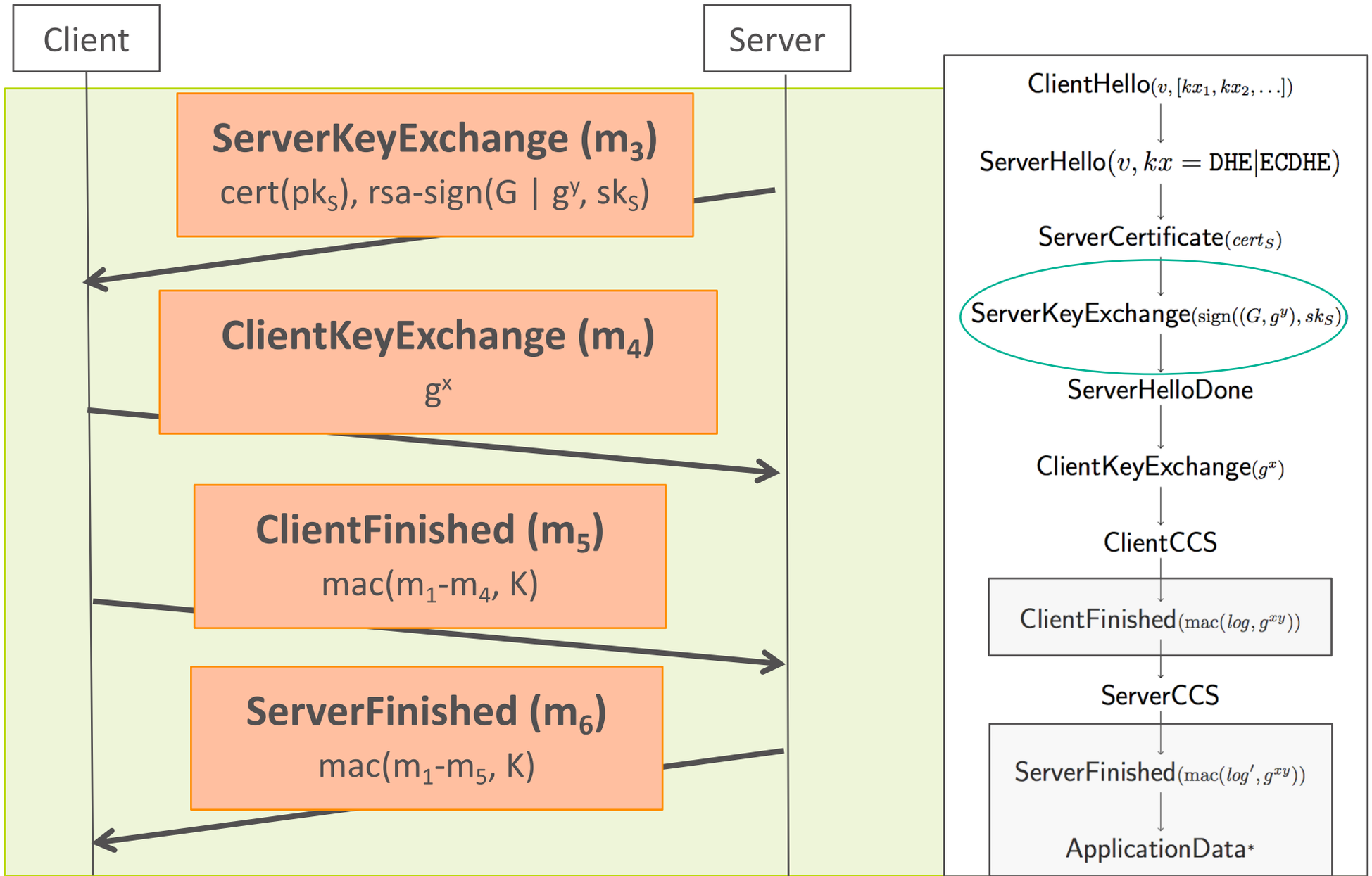
- AES-GCM, CBC MAC-Encode-Encrypt, RC4,…

100s of possible (obscure) protocol combinations!

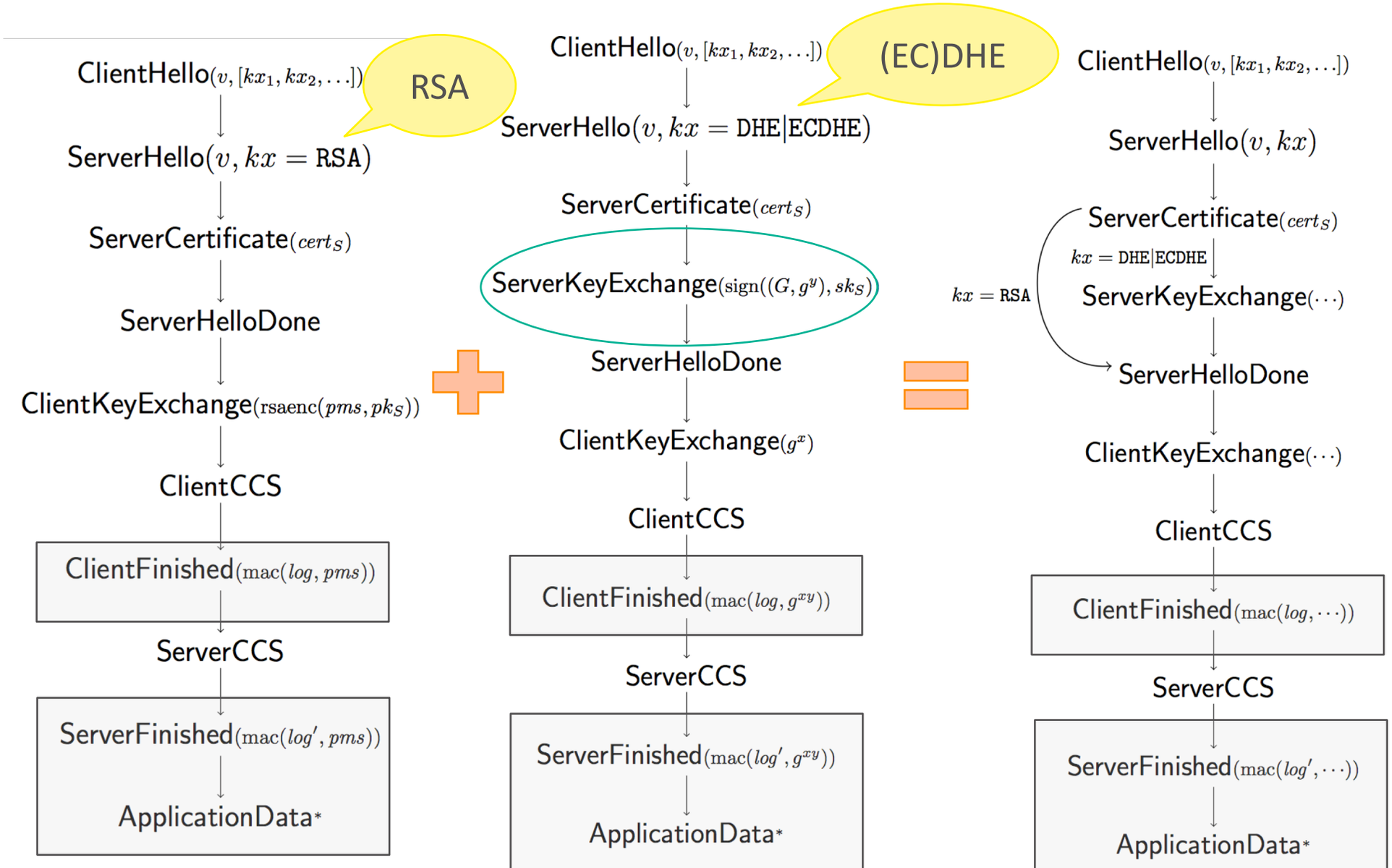
TLS 1.2 RSA Handshake



TLS 1.2 DHE Handshake



Composing RSA and DHE Handshakes



TLS State Machine

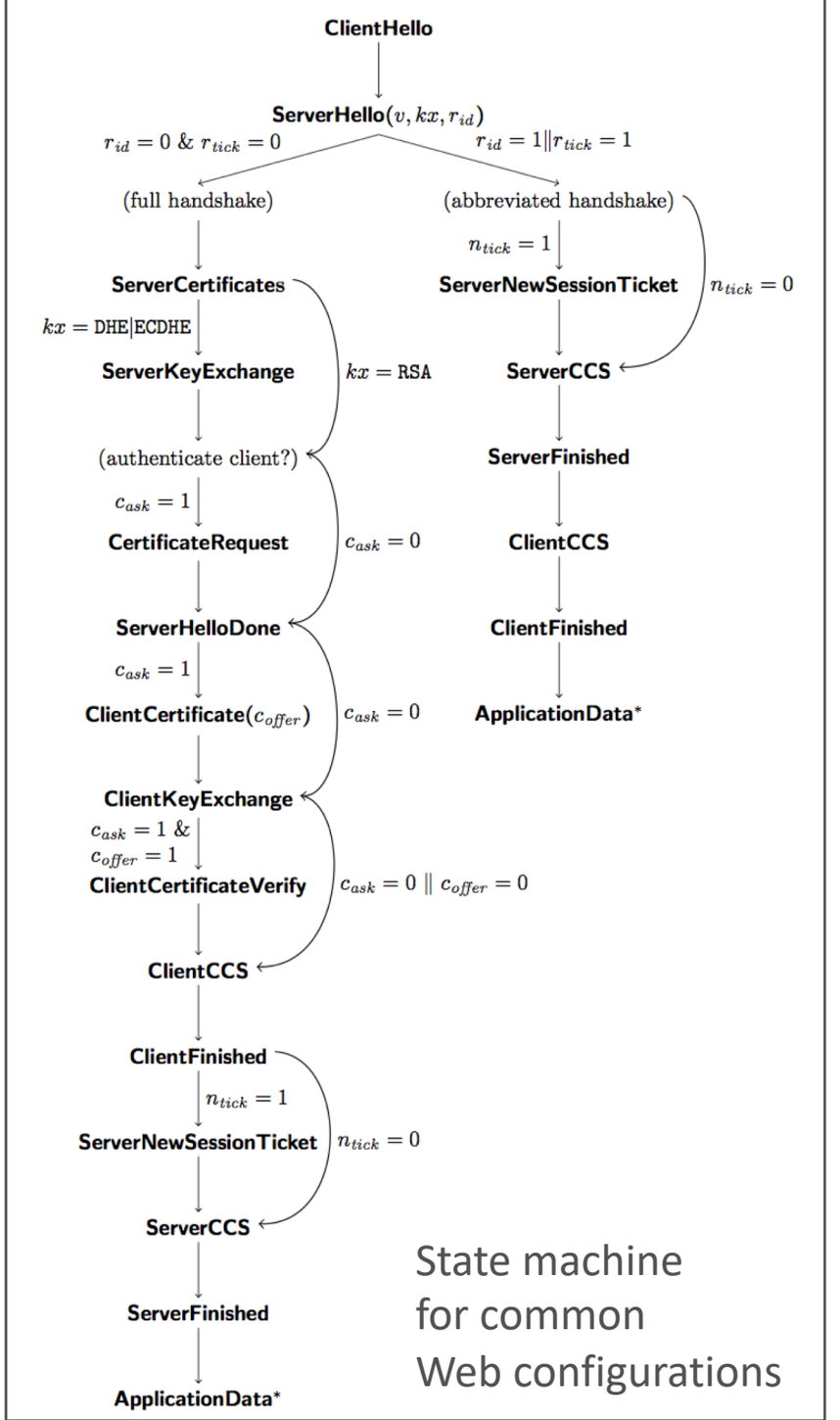
RSA + DHE + ECDHE

+ Session Resumption

+ Client Authentication

- Covers most features used on the Web
- Already quite a complex combination of protocols!

- State machine left unspecified in the RFC, so implementations are free to design their own



State machine for common Web configurations

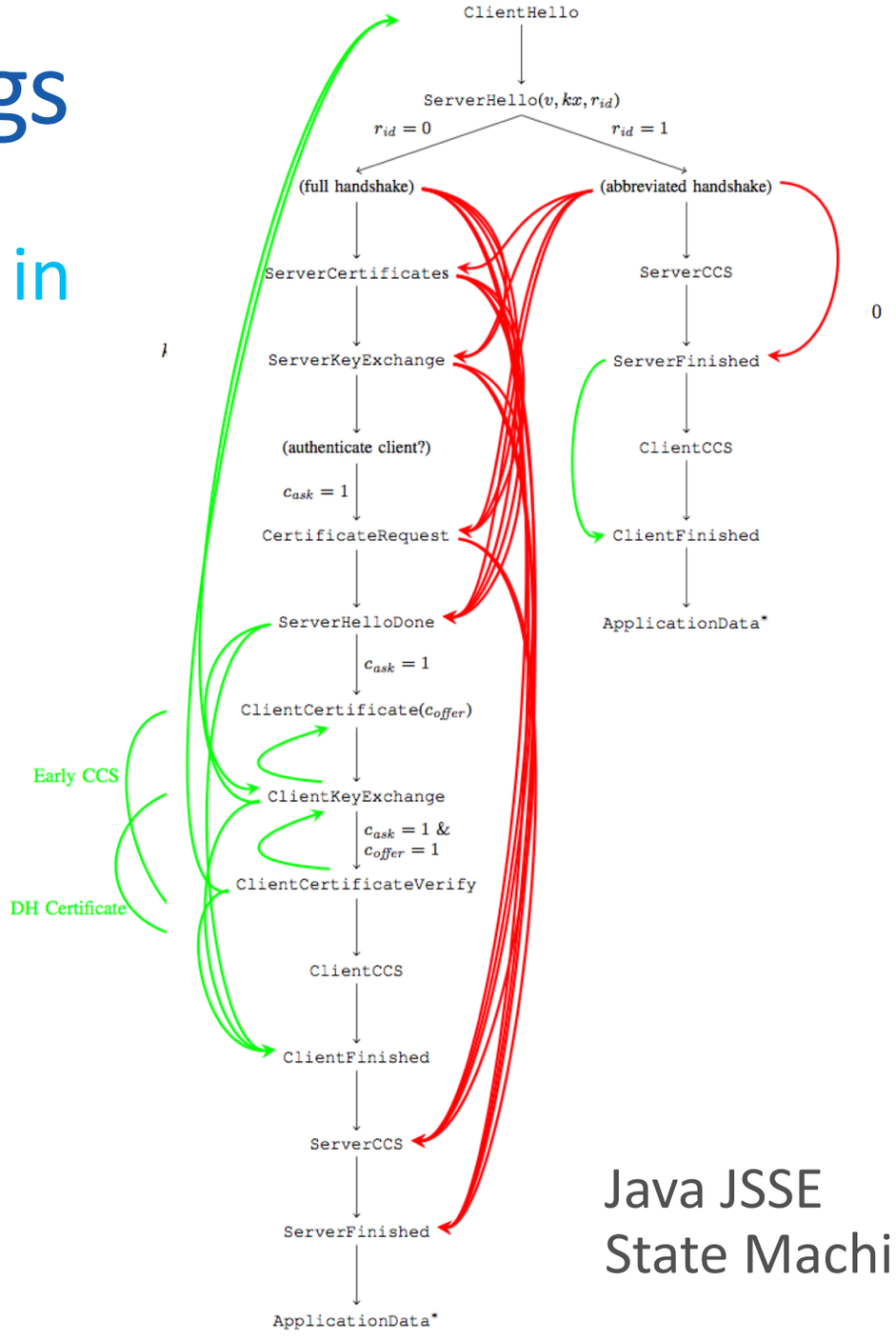
Many State Machine Bugs

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

How come all these bugs?

- In independent code bases, sitting in there for years
- Are they exploitable?

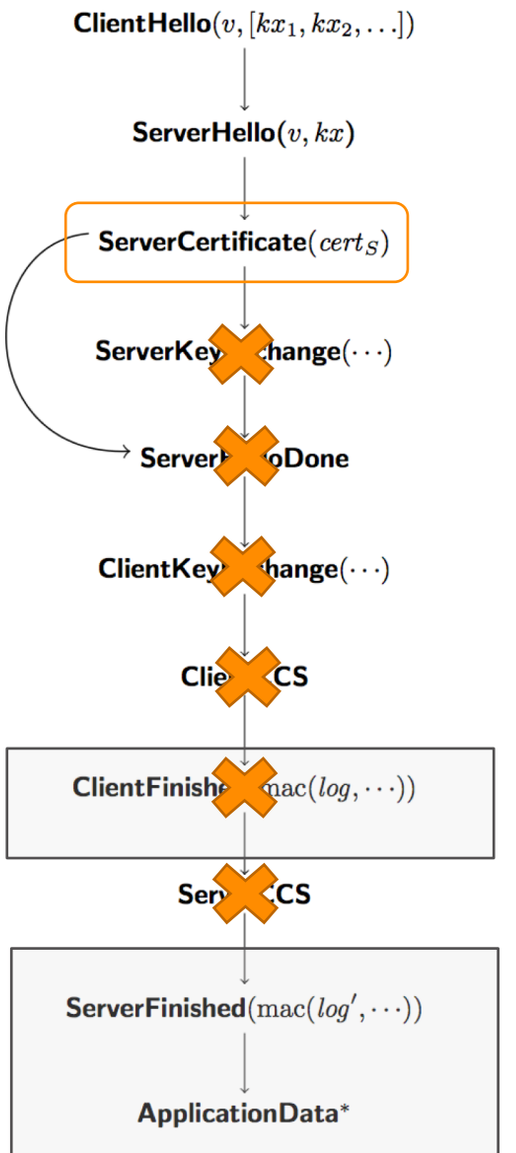


Java JSSE
State Machine

SKIPping Inconvenient Messages with Java

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



Impact of State Machine Bugs

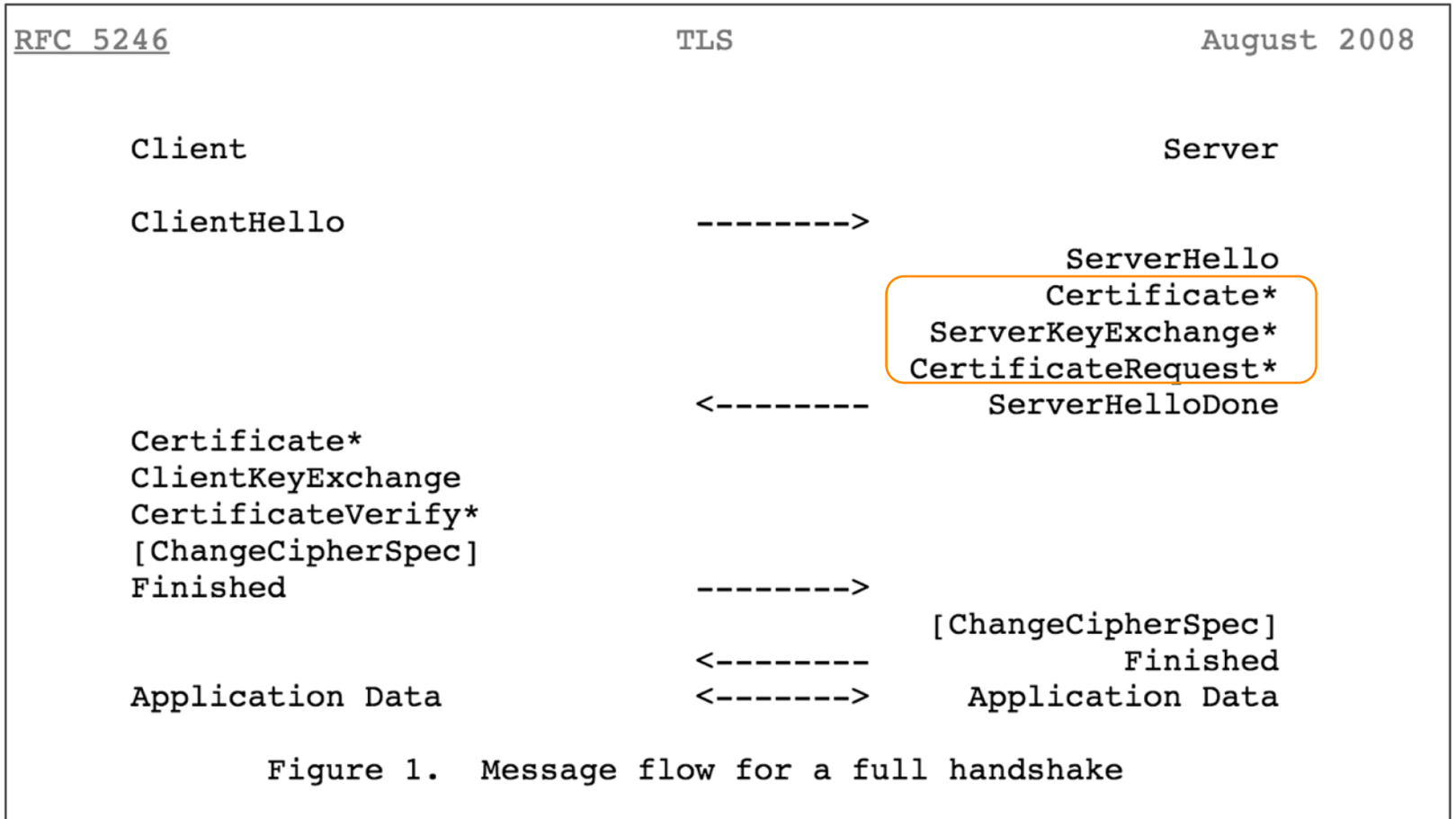
- **SKIP:** Until Jan 2015, Java SSL/TLS provided no protection against an active network attacker

- **FREAK:** OpenSSL, SecureTransport (Safari), SChannel (Internet Explorer) allowed confusions between 2048-bit RSA and 512-bit EXPORT RSA handshakes
  - In 2015, 35% of web servers supported EXPORT-grade cryptography
  - An active network attacker who can factor 512-bit RSA could impersonate 35% of the Web to most web browsers

Fix: Formally Specify and Verify the State Machine

TLS 1.2 specifies a ladder diagram with optional messages

- Relies on the Finished messages to ensure agreement



RFC 5246 TLS August 2008

```
      Client                                               Server

      ClientHello                  -------->
                                                      ServerHello
                                                     Certificate*
                                               ServerKeyExchange*
                                              CertificateRequest*
                                   <--------      ServerHelloDone
      Certificate*
      ClientKeyExchange
      CertificateVerify*
      [ChangeCipherSpec]
      Finished                     -------->
                                                [ChangeCipherSpec]
                                   <--------             Finished
      Application Data             <------->     Application Data
```

Figure 1. Message flow for a full handshake

Proving State Machine Code

Specify a state machine with deterministic transitions

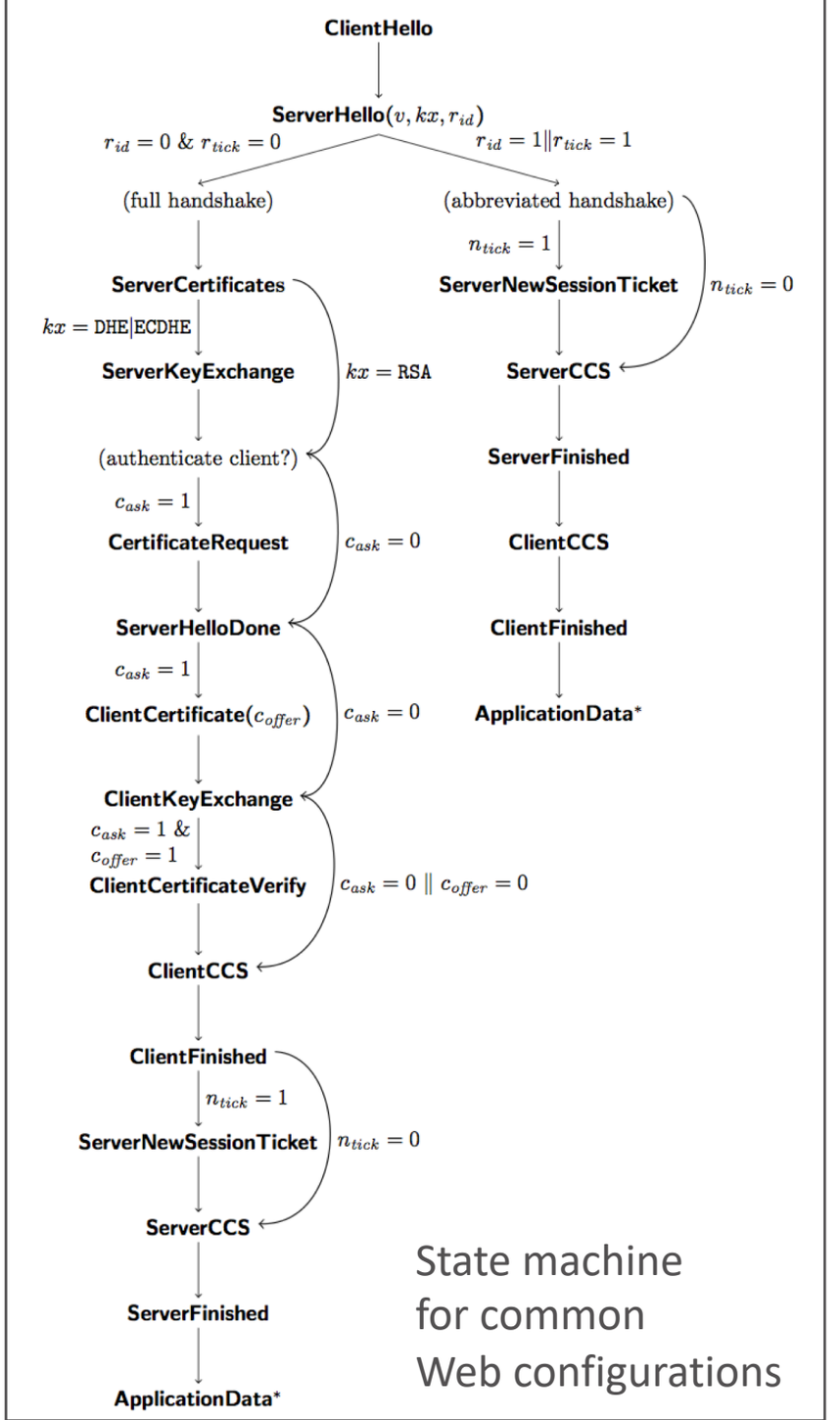
- Beware of transitions triggered by unauthenticated data!

Implement it in C/Rust/F* as an isolated module

- Don’t completely mix parsing, crypto, and state transitions.

Prove that the code implements the state machine

- C proofs using Frama-C,
  F* proofs by typing



State machine for common Web configurations

Formal Verification

Verified crypto protocol designs

- Symbolic analysis to find attacks [ProVerif, Tamarin]
- Cryptographic proofs of security [CryptoVerif, F*, EasyCrypt]

Verified crypto protocol software

- Verified crypto libraries [F*, Coq, VST, Cryptol/SAW]
- Verified protocol implementations [F*, FCF/Coq]

Verifying Crypto and Protocol Code:

The Everest Verification Toolchain

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

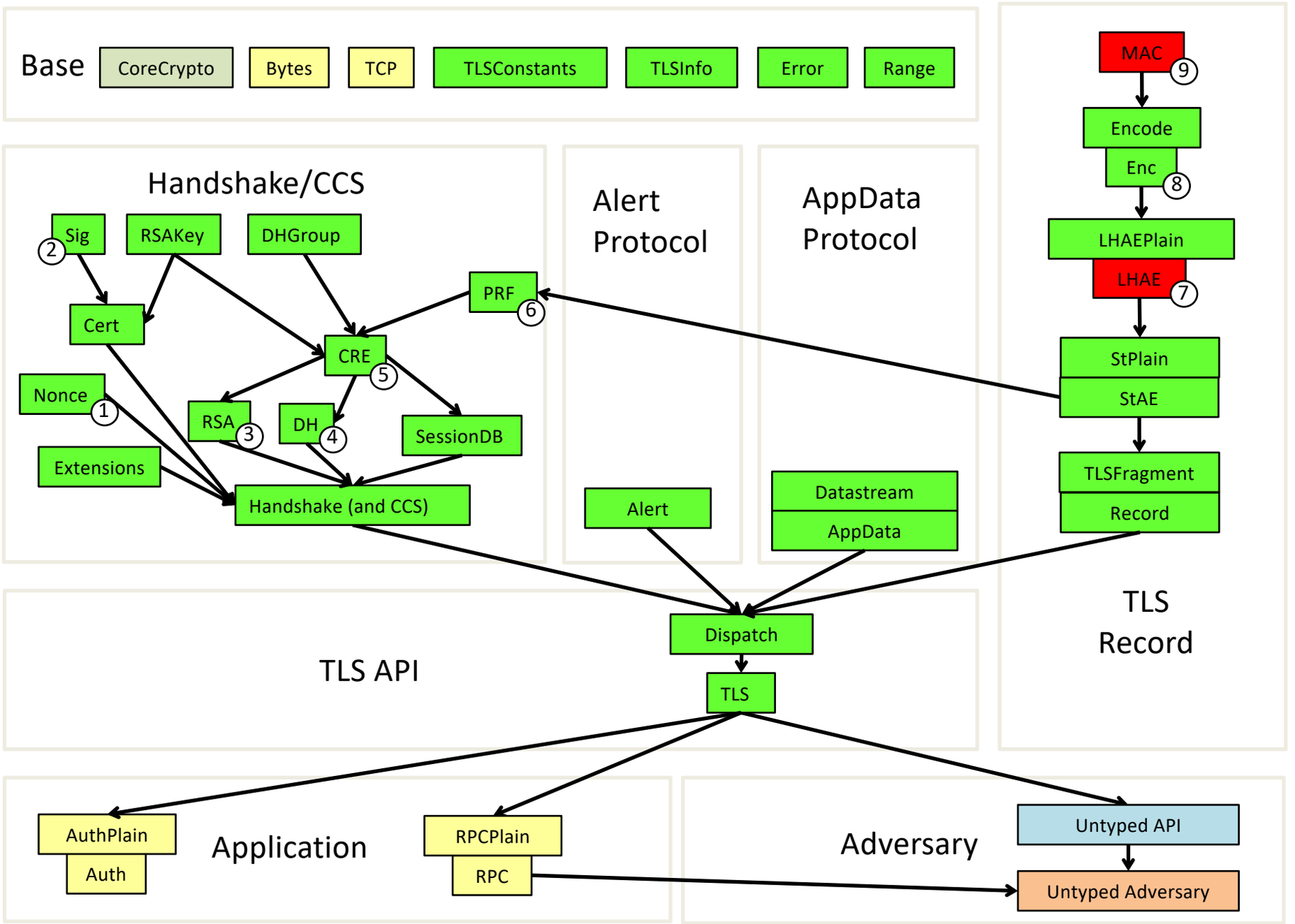
C/C++

**gcc, compcert,
clang, msvc**

interop with rest of
TLS/HTTPS ecosystem

production code

miTLS: Verified Implementation of TLS



Verified Crypto Software:
Implementing Fast, Verified Modern Crypto in HACL*

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

Modular Arithmetic in Crypto

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

$0 < x,y < p$
g fixed

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

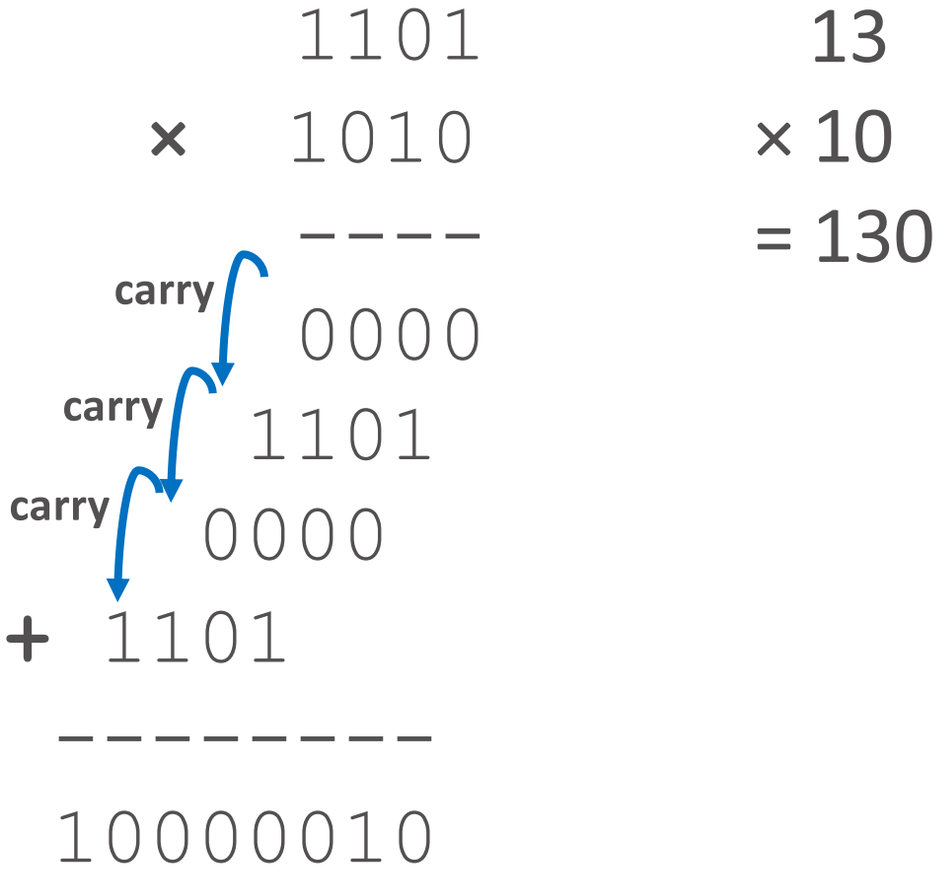
Implementing Modular Exponentiation

$$a^b \mod n$$

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

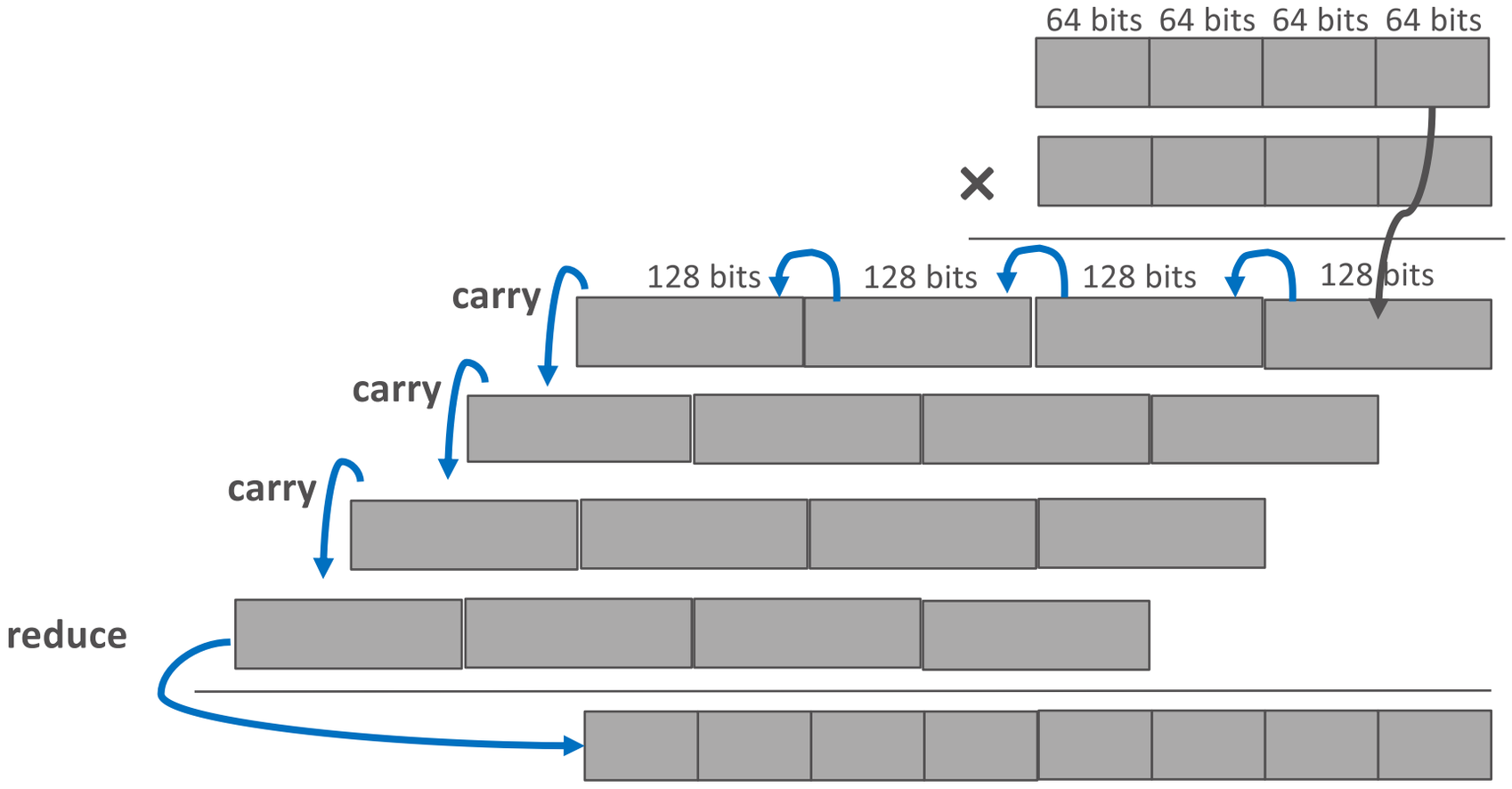
Big Integer (≥ 256-bit) arithmetic

Textbook Multiplication



13
× 10
= 130

256-bit Modular Multiplication on 64-bit Computers



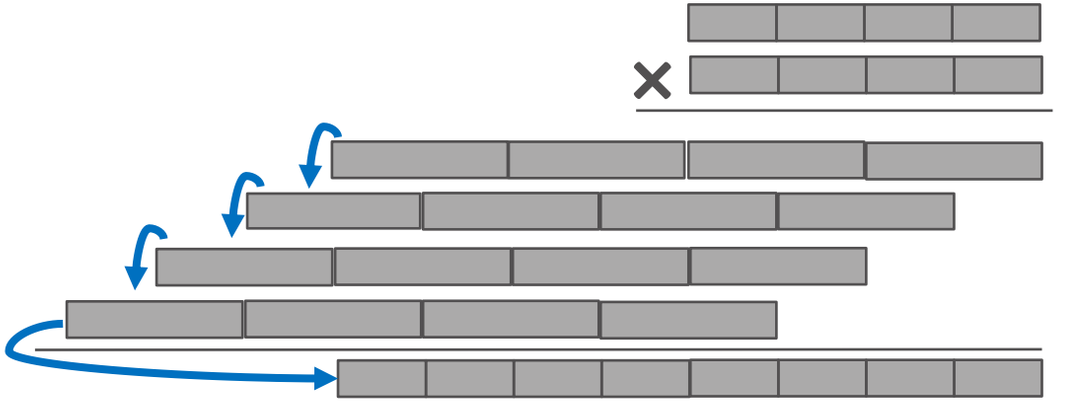
256-bit Modular Multiplication on 64-bit Computers

What can go wrong?

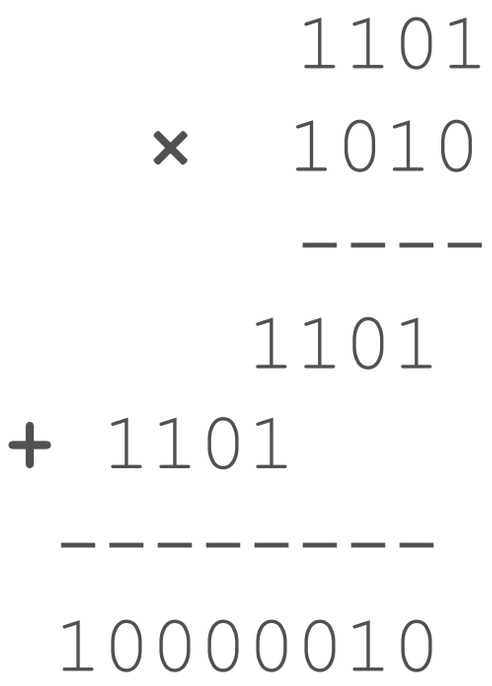
- Integer overflow
  (undefined output)
- Buffer overflow/underflow
  (memory error)
- Missing carry steps
  (wrong answer)
- Side-channel Attack
  (leaks secrets)

How expensive is it?

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



Optimizations vs. Side-Channel Leaks



Skipping steps is faster

- Fewer additions, carries

… but may leak information

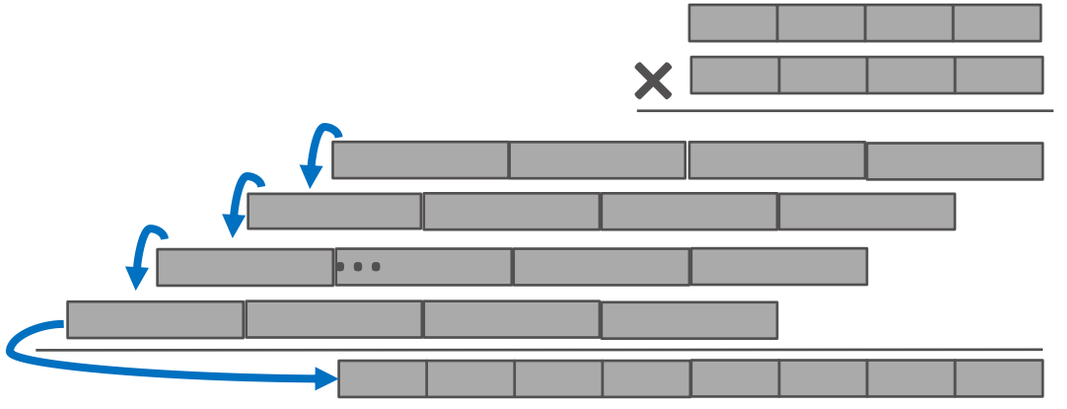
- Runtime proportional to number of 1s in 1010
- Attacker can observe runtime to guess input
- Input may be secret key!

Optimizations in Modular Arithmetic

Many prime-specific optimizations

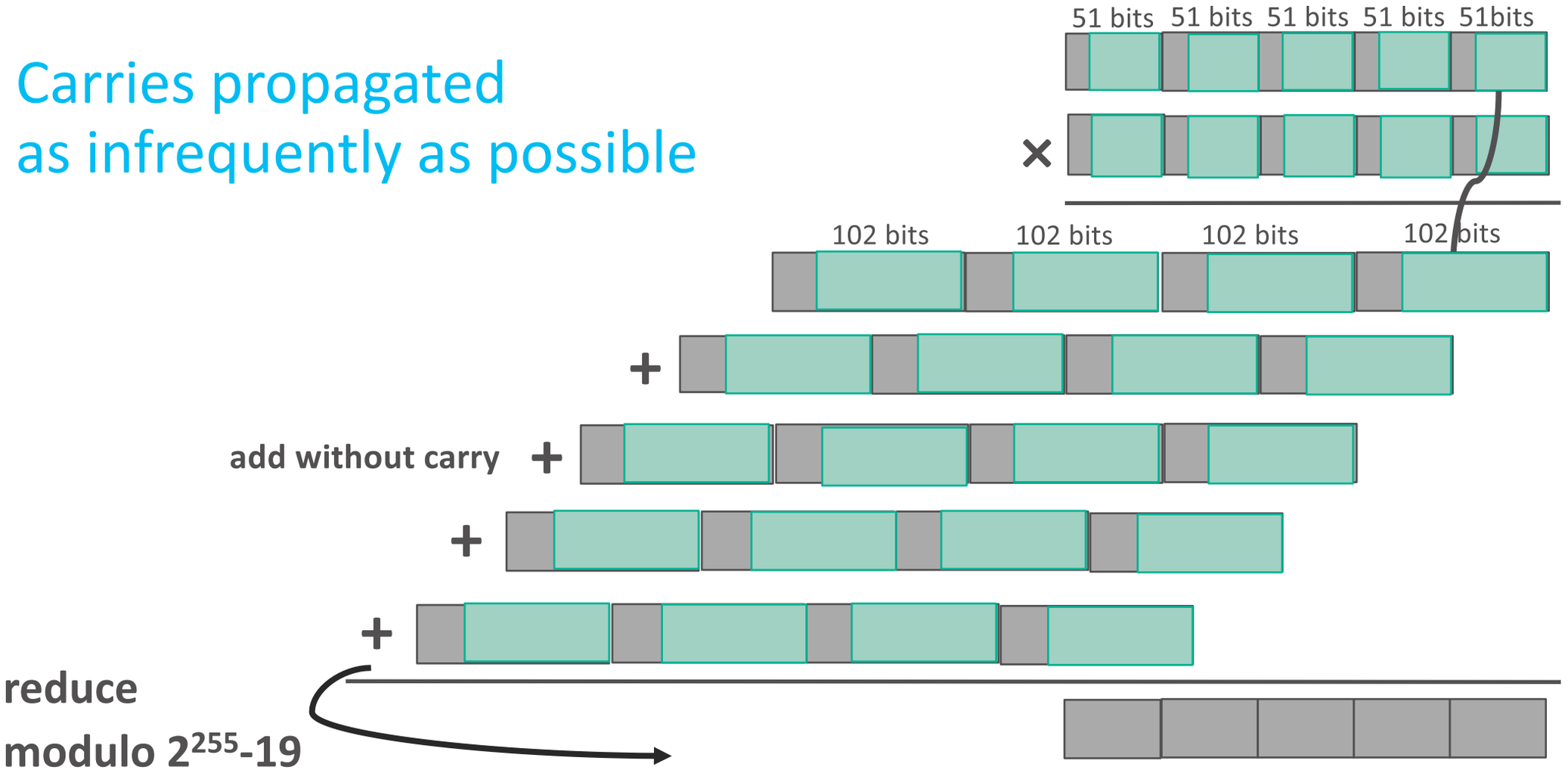
- Trade-off multiplication vs modular reduction
- Use only 51 out of 64 bits to avoid carries
- Precompute reusable intermediate values
- Parallelize (vectorize) multiplication and squaring

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, ...

Towards High-Assurance Crypto Software

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]

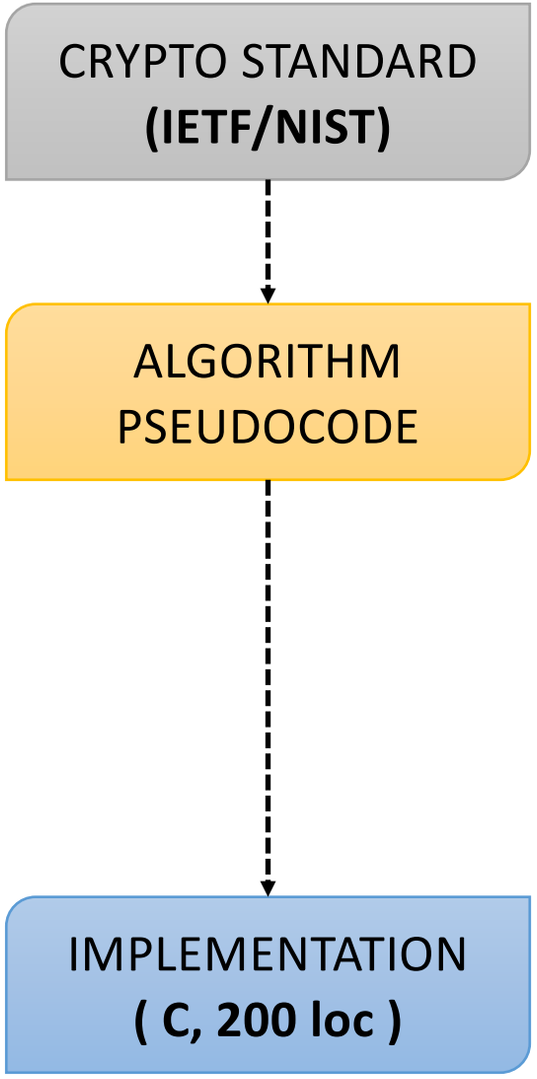
Formal verification can systematically prevent bugs

- *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

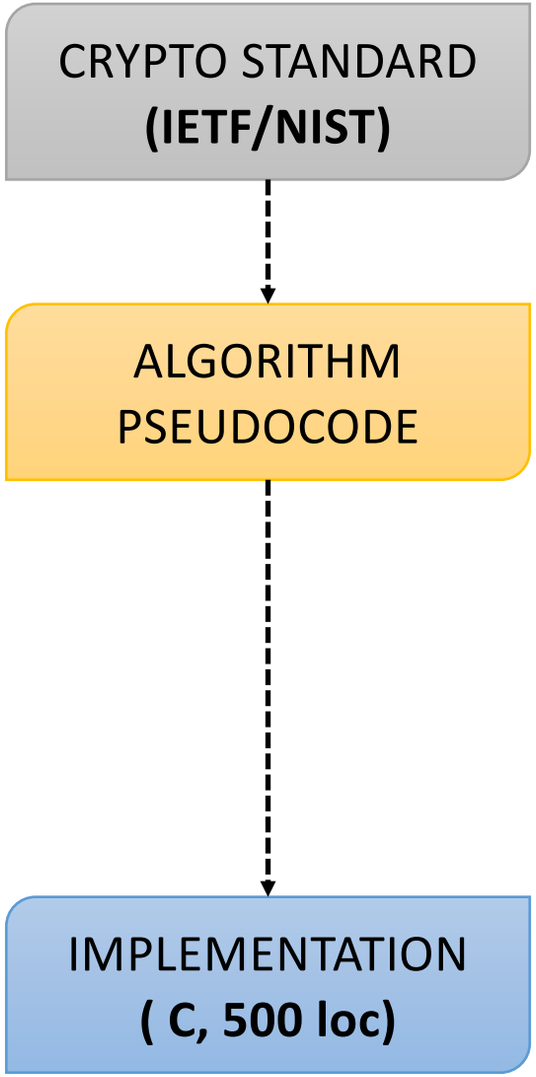



```
static void chacha20_core(chacha_buf *output, const u32 input[16])
{
    u32 x[16];
    int i;
    const union {
        long one;
        char little;
    } is_endian = { 1 };

    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



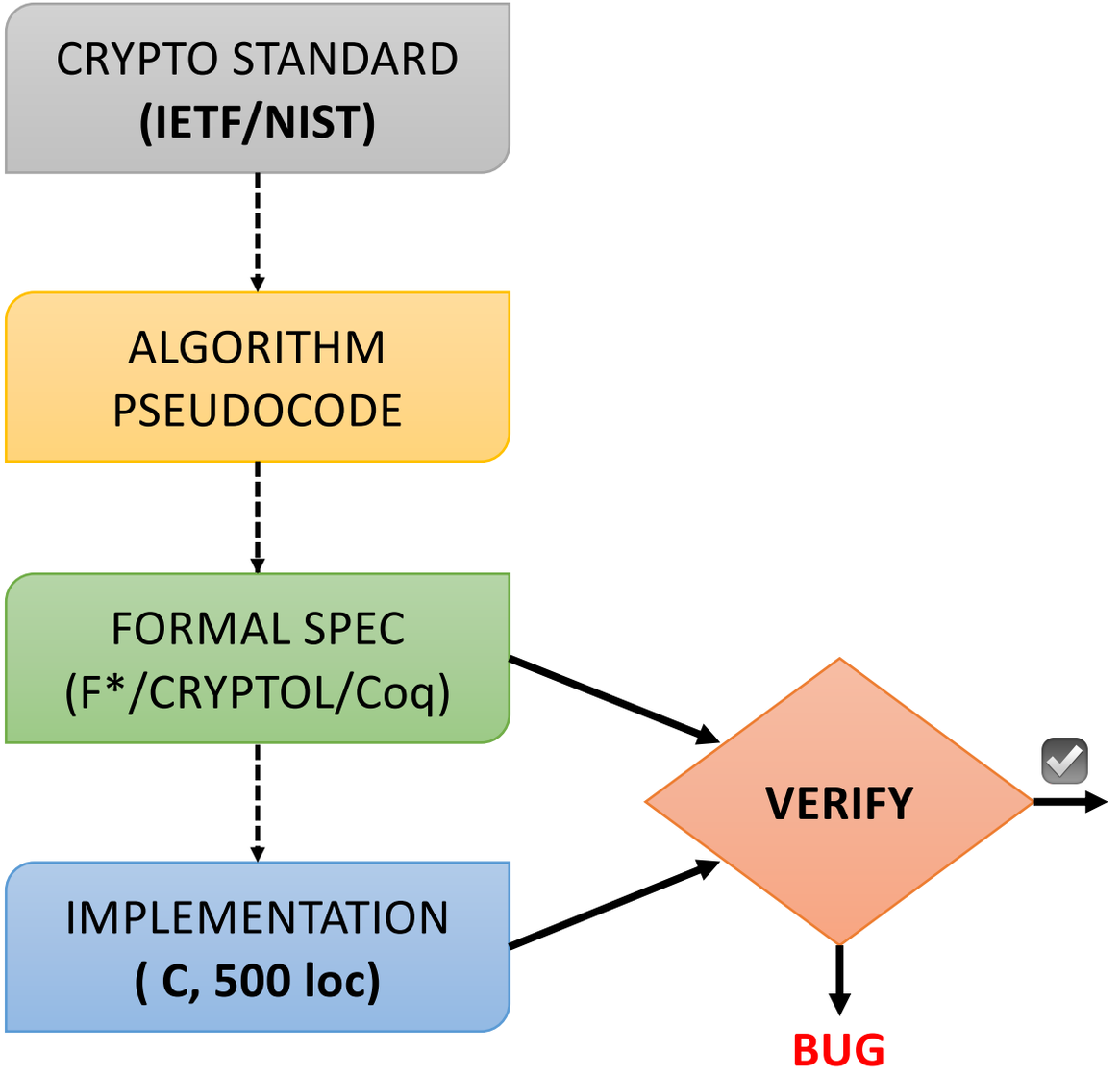
2.

```
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);
```

Optimized 32-bit Code

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

F*: a verification oriented language

F* features:

- Functional language (« à la Ocaml »)
- Customizable verification system (« à la Coq »)
- Proof automation via SMT solvers (Z3)
- Compilation tools to C, WebAssembly, OCaml

http://fstar-lang.org

Example: Poly1305 MAC Algorithm

Poly1305 is a message authentication code

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

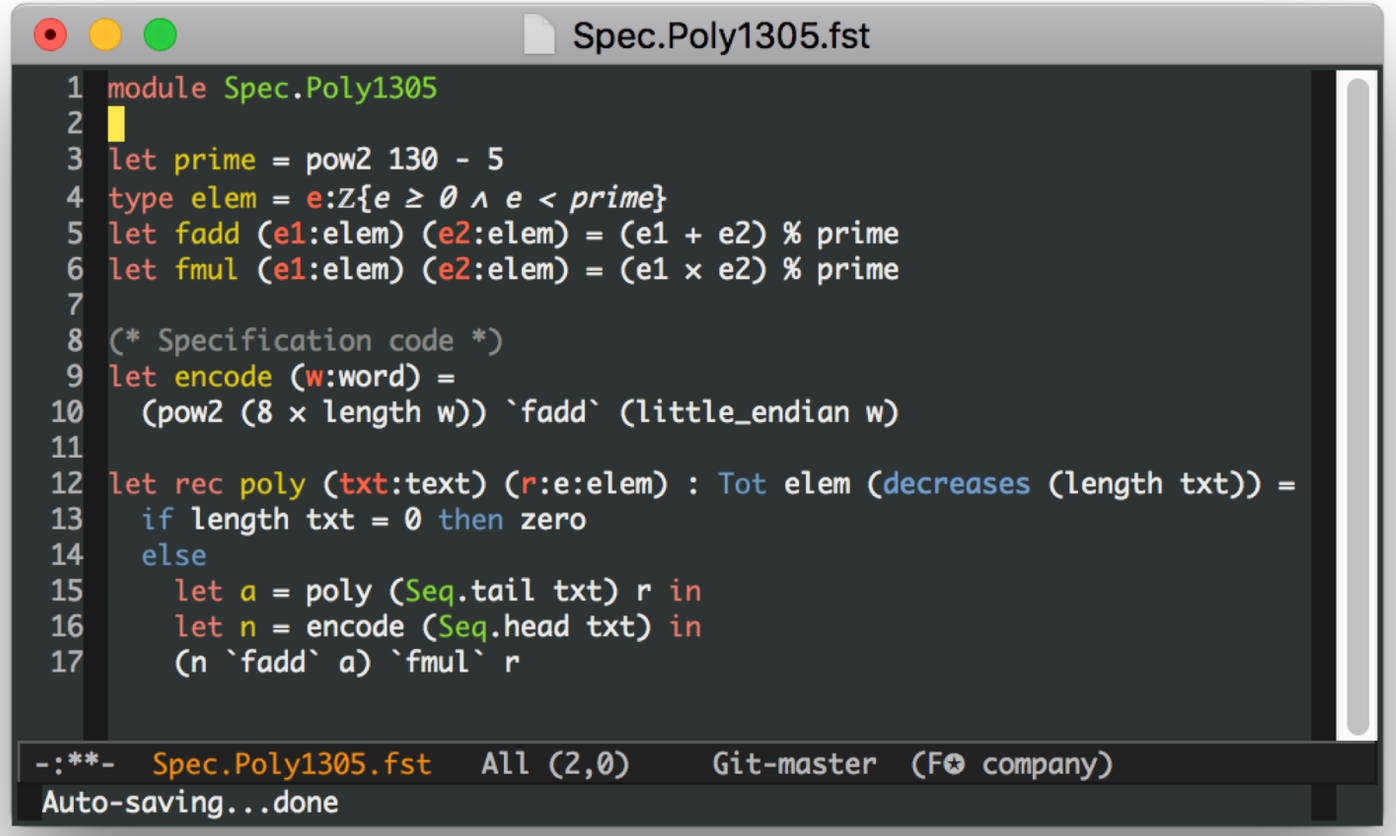
It authenticates a data stream $w_1 \ldots w_n$ by

- 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



Implementing Poly1305 in Stateful F*

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

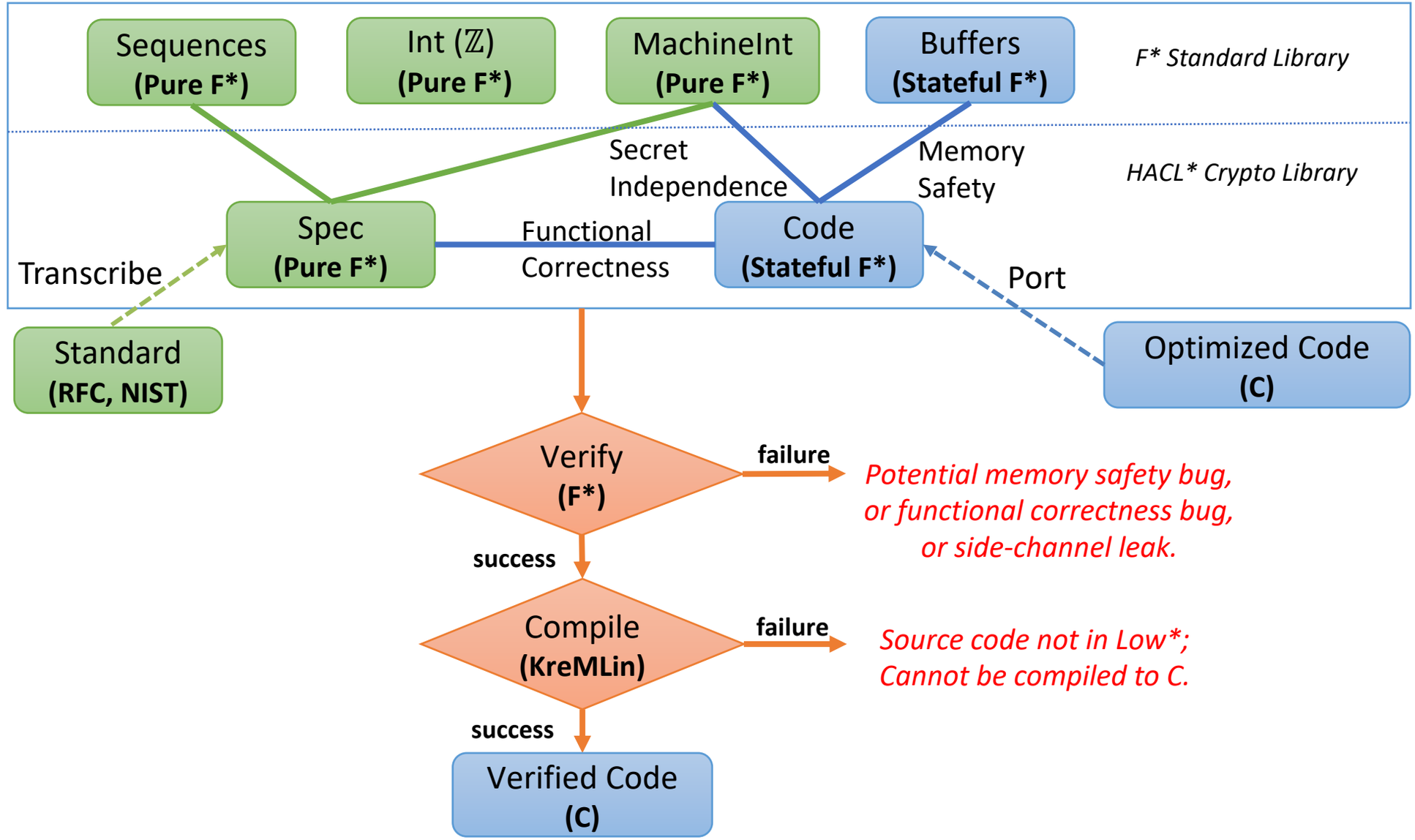
Verifying Secret Independence

```
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

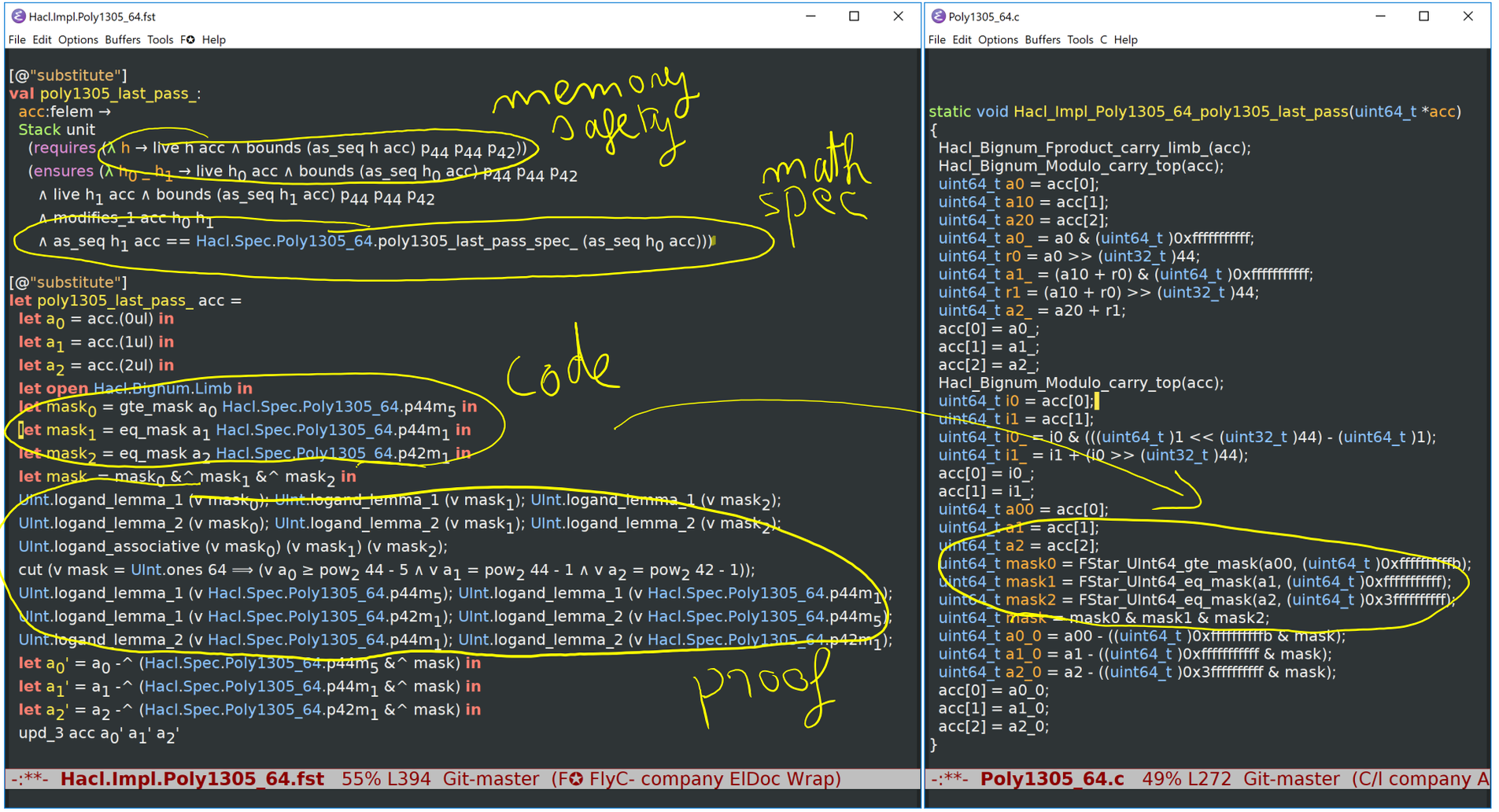
  (Forbidden operations depend on target platform)

HACL* Verification Workflow



F* Code for Poly1305

Compiled C Code



Verifying Curve25519 Code:
F* Demo

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**

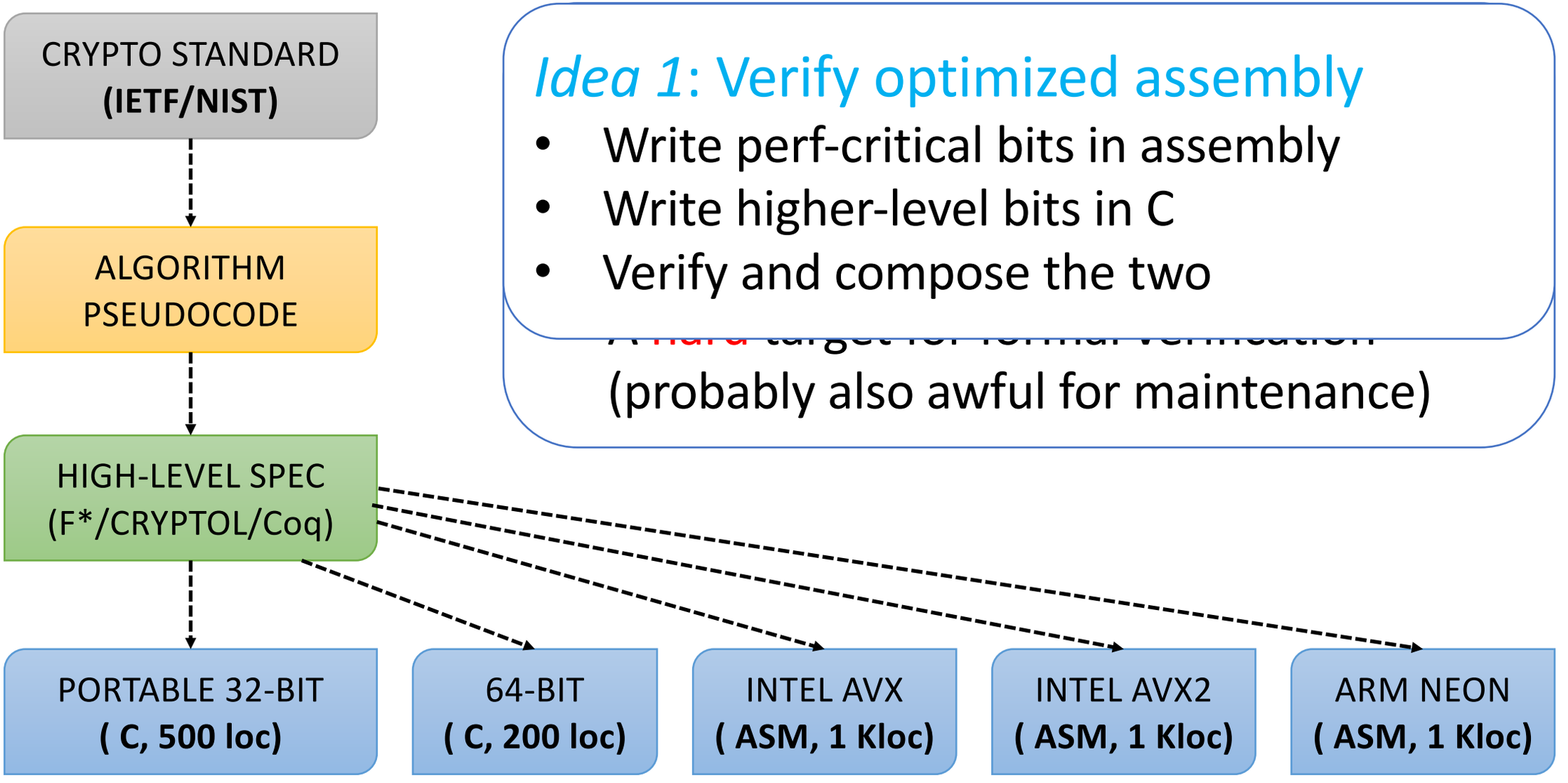
POLY1305

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

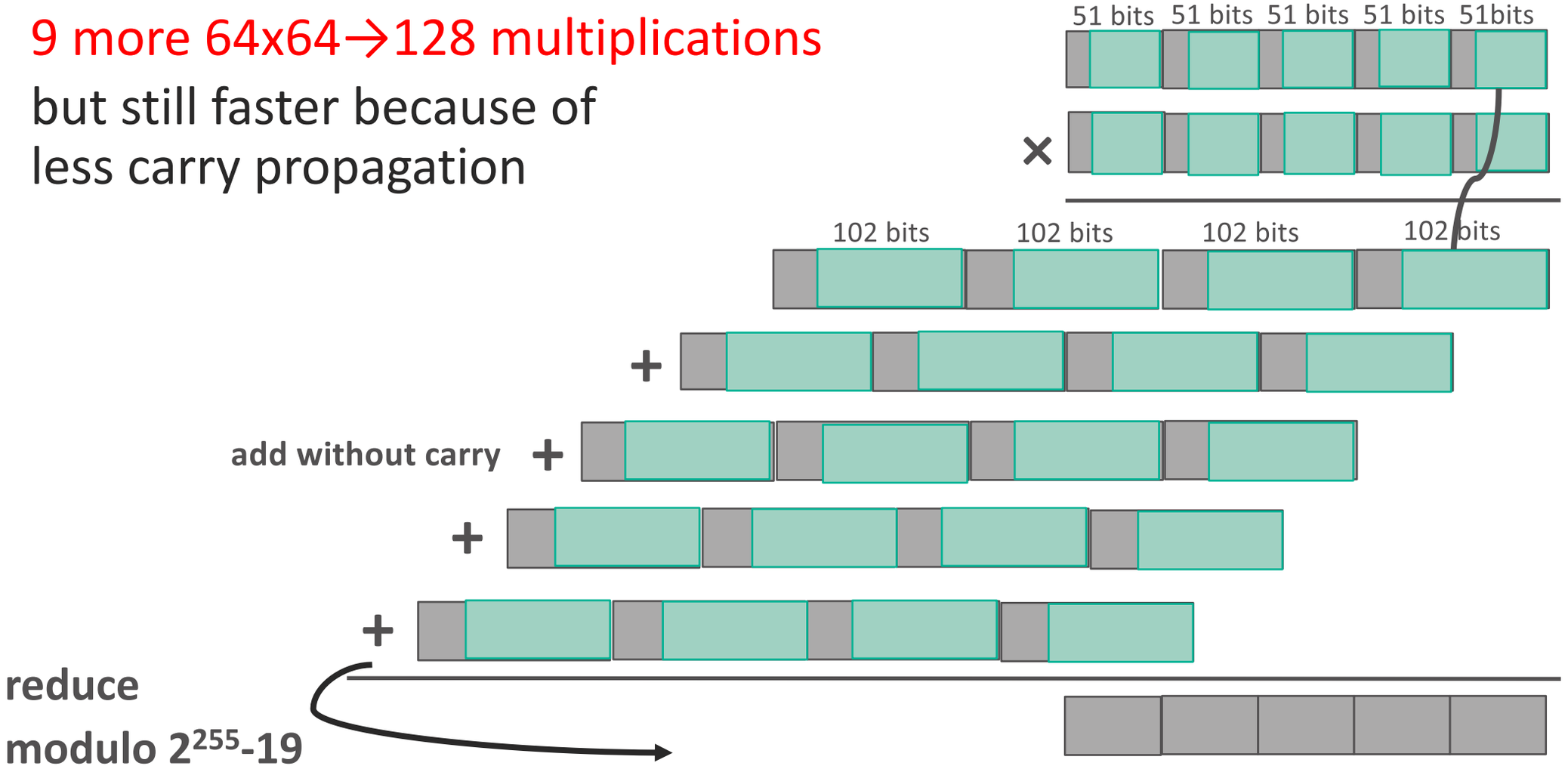
- Write perf-critical bits in assembly
- Write higher-level bits in C
- Verify and compose the two

A hard target for formal verification
(probably also awful for maintenance)

Recall: Unsaturated Arithmetic for Curve25519

9 more 64x64→128 multiplications

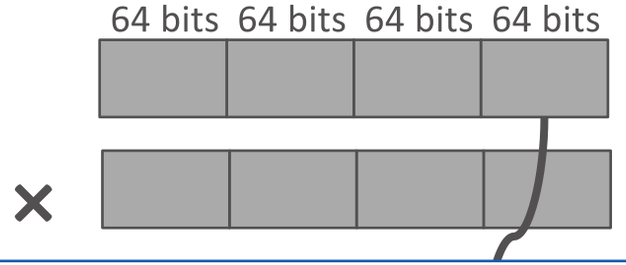
but still faster because of
less carry propagation



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.

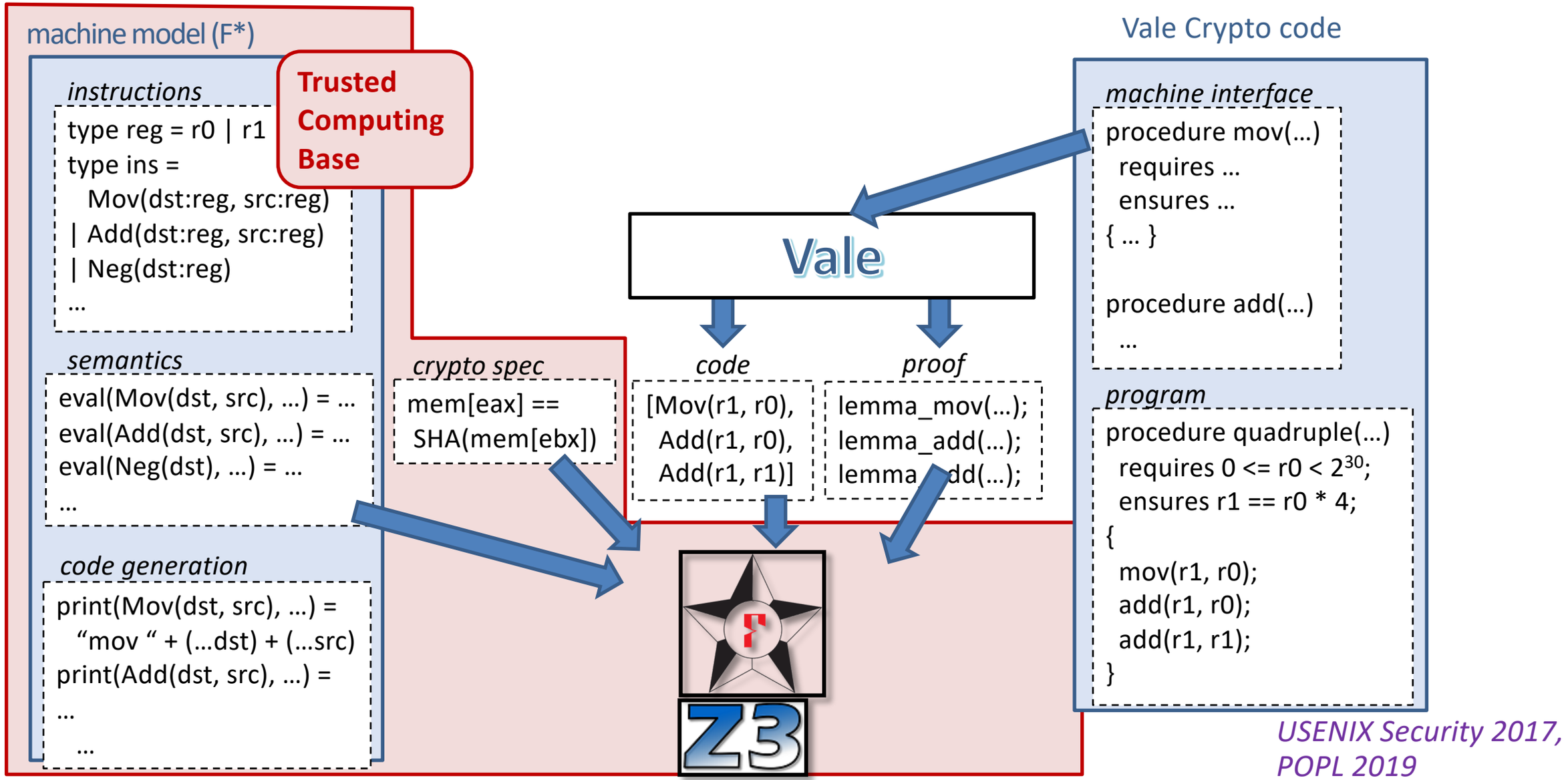
> *Post by Jason A. Donenfeld*
> - It looks like the precomputation implementation is failing some unit tests! Perhaps it's not properly reducing incoming public points? There's the vector if you'd like to play with it. The other test vectors I have do pass, though, which is good I suppose.

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

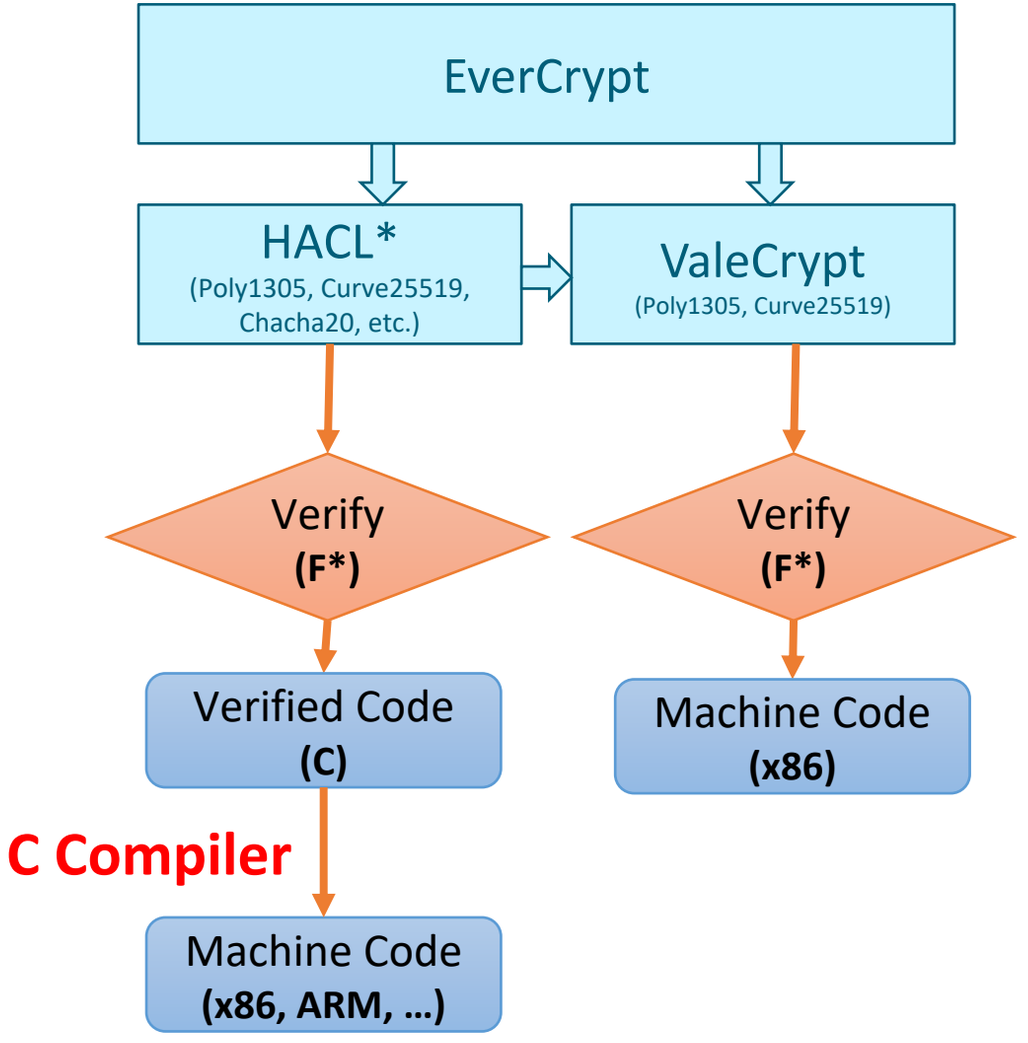
Implement curve operations in HACL*

- 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*



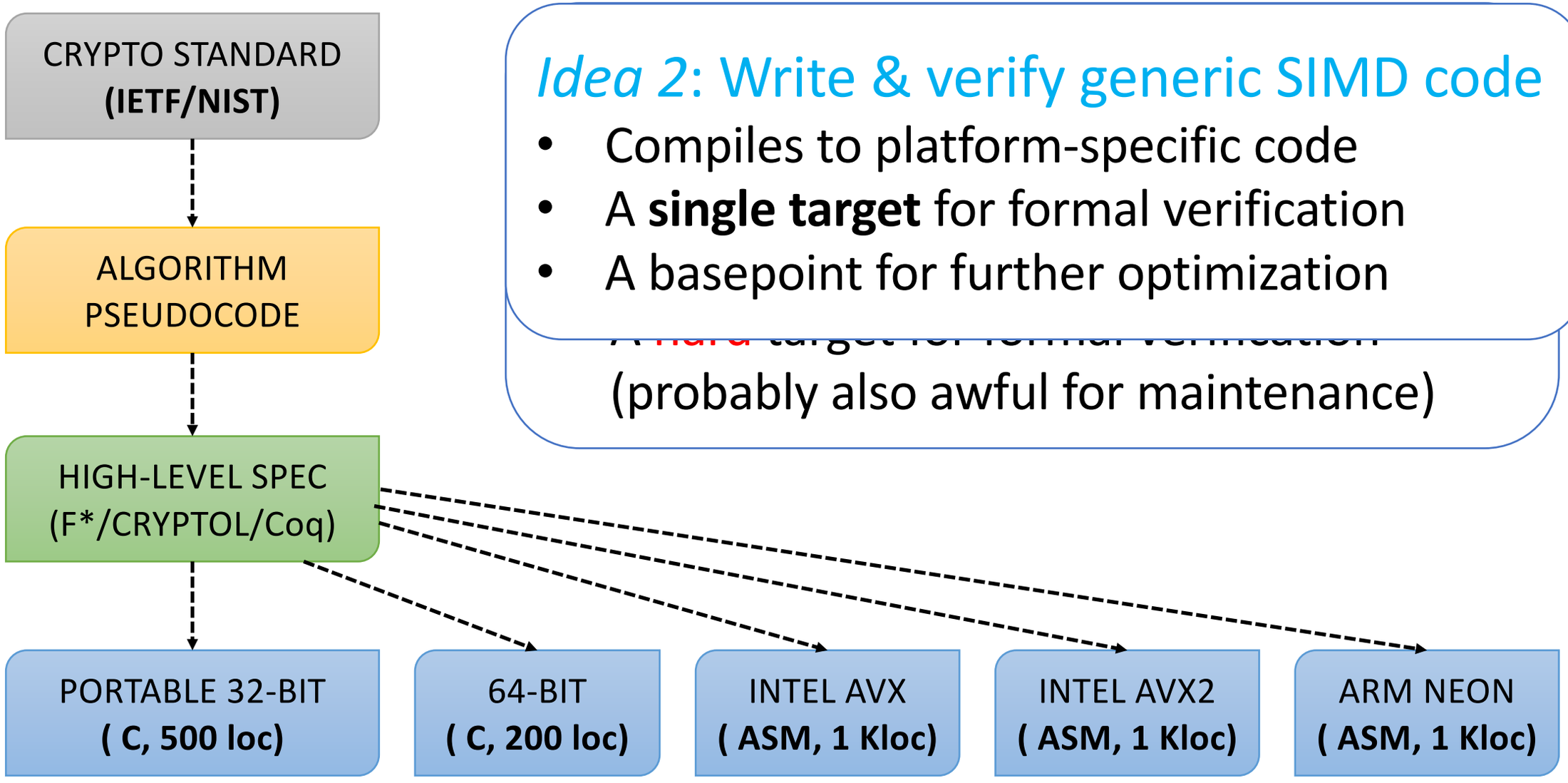
Algorithm	C version	Targeted ASM version
AEAD		
AES-GCM		AES-NI + PCLMULQDQ + AVX
ChachaPoly	yes	
Hashes		
MD5	yes	
SHA1	yes	
SHA2	yes	SHA-EXT (for SHA2-224+SHA2-256)
MACS		
HMAC	yes	agile over hash
Poly1305	yes	X64
Key Derivation		
HKDF	yes	agile over hash
ECC		
Curve25519	yes	BMI2 + ADX
Ciphers		
Chacha20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

A hard target for formal verification
(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

Concluding Thoughts

Building high-assurance crypto is a collaborative process

- Verification research has made advances, but we need help

If you are a cryptographer: try writing formal specs for your fancy new primitive or protocol

- Use ProVerif, or Cryptol, or Coq, or F*, or … try hacspec

If you are a crypto developer: consider writing generic code with clearly stated pre-/post-conditions

- Don’t just add yet more undocumented unverified assembly

Questions?

- ProVerif: http://proverif.inria.fr
- Cryptoverif: http://cryptoverif.inria.fr
- F*: http://www.fstar-lang.org/

- Project Everest: https://project-everest.github.io/
- HACL*: https://github.com/project-everest/hacl-star
- hacspec: https://github.com/HACS-workshop/hacspec