Verified Vectorized Cryptography

(with less manual effort)

Karthikeyan Bhargavan

Inria

+

B. Beurdouche, M. Polubelova, N. Kulatova

J. Protzenko, S. Zanella-Béguelin

Inria

Microsoft

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]

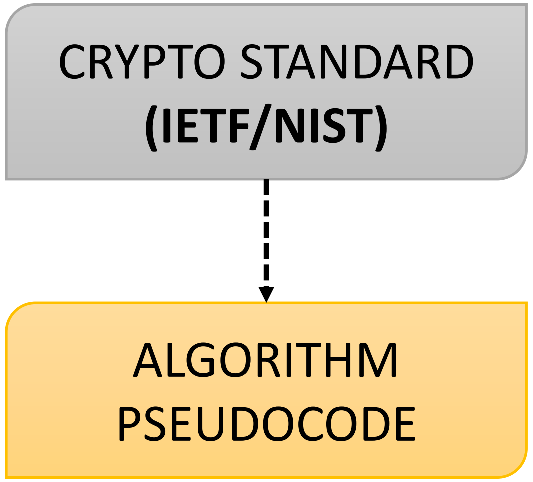
Formal verification can systematically prevent bugs

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

How do we scale verification up to full crypto libraries?

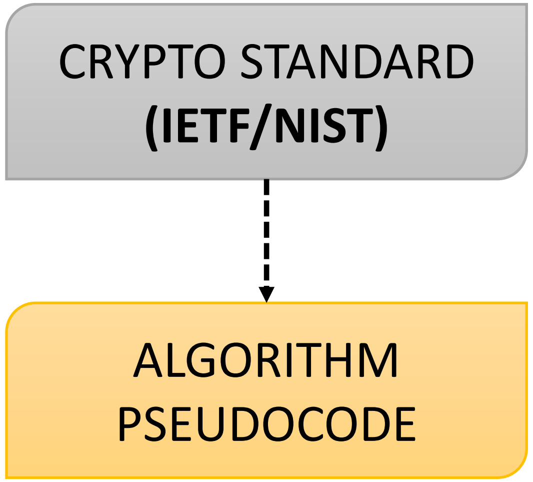
- Low-level platform specific optimizations for a suite of algorithms

Writing Verified Crypto Code




```
Obsoleted by: 8439                                              INFORMATIONAL
                                                                Errata Exist
Internet Research Task Force (IRTF)                                   Y. Nir
Request for Comments: 7539                                       Check Point
Category: Informational                                          A. Langley
ISSN: 2070-1721                                                 Google, Inc.
                                                                    May 2015


                ChaCha20 and Poly1305 for IETF Protocols

Abstract

   This document defines the ChaCha20 stream cipher as well as the use
   of the Poly1305 authenticator, both as stand-alone algorithms and as
   a "combined mode", or Authenticated Encryption with Associated Data
   (AEAD) algorithm.

   This document does not introduce any new crypto, but is meant to
   serve as a stable reference and an implementation guide.  It is a
   product of the Crypto Forum Research Group (CFRG).
```

Writing Verified Crypto Code

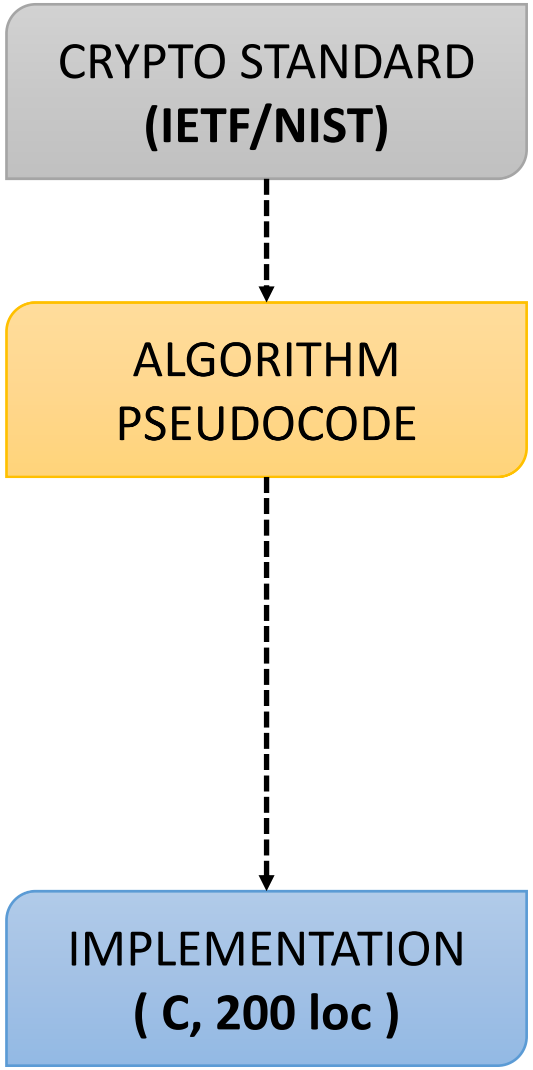


2.3.1. The ChaCha20 Block Function in Pseudocode

Note: This section and a few others contain pseudocode for the algorithm explained in a previous section. Every effort was made for the pseudocode to accurately reflect the algorithm as described in the preceding section. If a conflict is still present, the textual explanation and the test vectors are normative.

```
inner_block (state):
   Qround(state, 0, 4, 8,12)
   Qround(state, 1, 5, 9,13)
   Qround(state, 2, 6,10,14)
   Qround(state, 3, 7,11,15)
   Qround(state, 0, 5,10,15)
   Qround(state, 1, 6,11,12)
   Qround(state, 2, 7, 8,13)
   Qround(state, 3, 4, 9,14)
   end

chacha20_block(key, counter, nonce):
   state = constants | key | counter | nonce
   working_state = state
   for i=1 upto 10
      inner_block(working_state)
      end
   state += working_state
   return serialize(state)
   end
```

Writing Verified Crypto Code

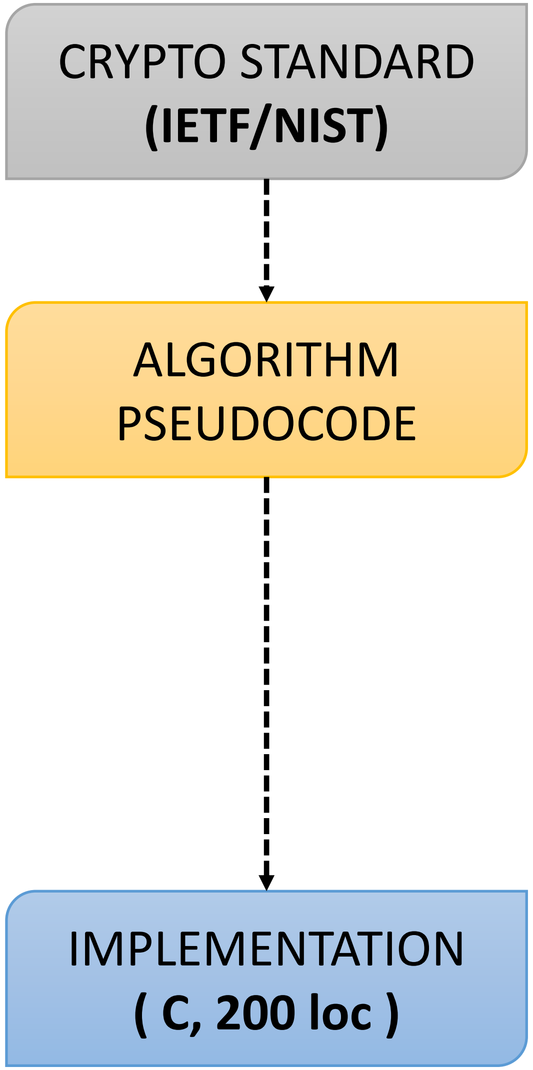



```c
static void chacha20_core(chacha_buf *output, const u32 input[16])
{
    u32 x[16];
    int i;
    const union {
        long one;
        char little;
    } is_endian = { 1 };

    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);
```

Adds many details

- Memory allocation
- Incremental API

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);
```

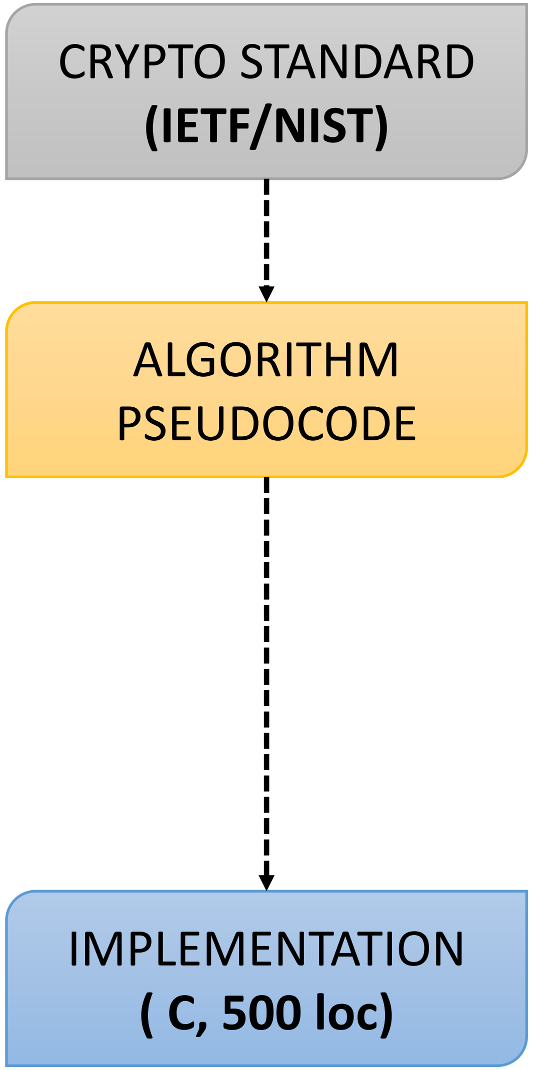
Adds many details

- Memory allocation
- Incremental API

Obviously correct?

unless we introduced a buffer overflow, or a timing leak

Writing Verified Crypto Code



2.5.1. The Poly1305 Algorithms in Pseudocode

```
clamp(r): r &= 0x0ffffffc0ffffffc0ffffffc0fffffff
poly1305_mac(msg, key):
   r = (le_bytes_to_num(key[0..15])
   clamp(r)
   s = le_num(key[16..31]
   accumulator = 0
   p = (1<<130)-5
   for i=1 upto ceil(msg
      n = le_bytes_to_nu
      a += n
      a = (r * a) % p
      end
   a += s
   return num_to_16_le_bytes(a)
   end
```

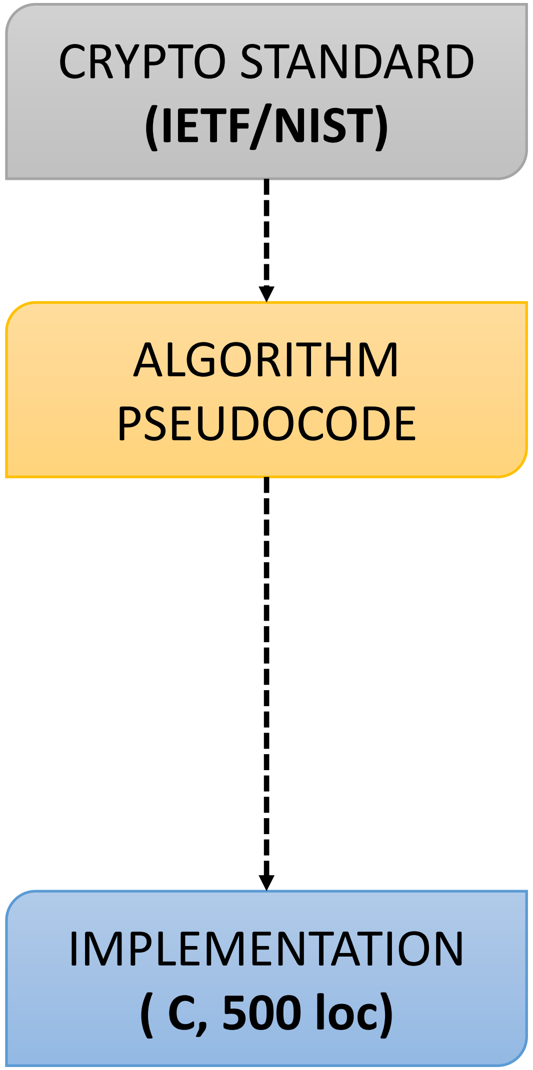
Modular Field Arithmetic

`a += n`

`a = (r*a) % (2^130 – 5)`

$$a = (r*a) \ \% \ (2^{130} - 5)$$

Writing Verified Crypto Code

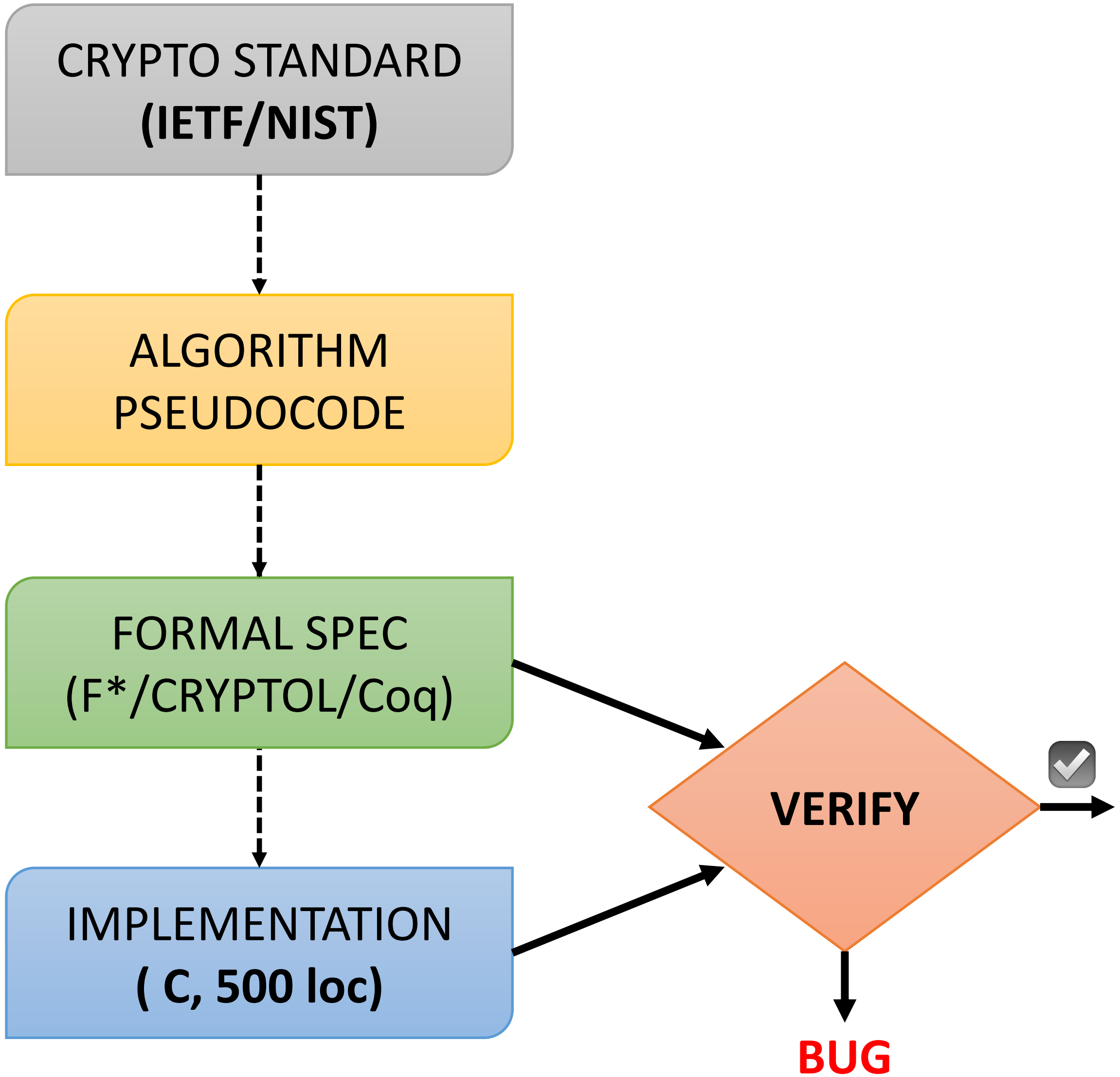



```
while (len >= POLY1305_BLOCK_SIZE) {
    /* h += m[i] */
    h0 = (u32)(d0 = (u64)h0 + U8TOU32(inp + 0));
    h1 = (u32)(d1 = (u64)h1 + (d0 >> 32) + U8TOU32(inp + 4));
    h2 = (u32)(d2 = (u64)h2 + (d1 >> 32) + U8TOU32(inp + 8));
    h3 = (u32)(d3 = (u64)h3 + (d2 >> 32) + U8TOU32(inp + 12));
   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

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**

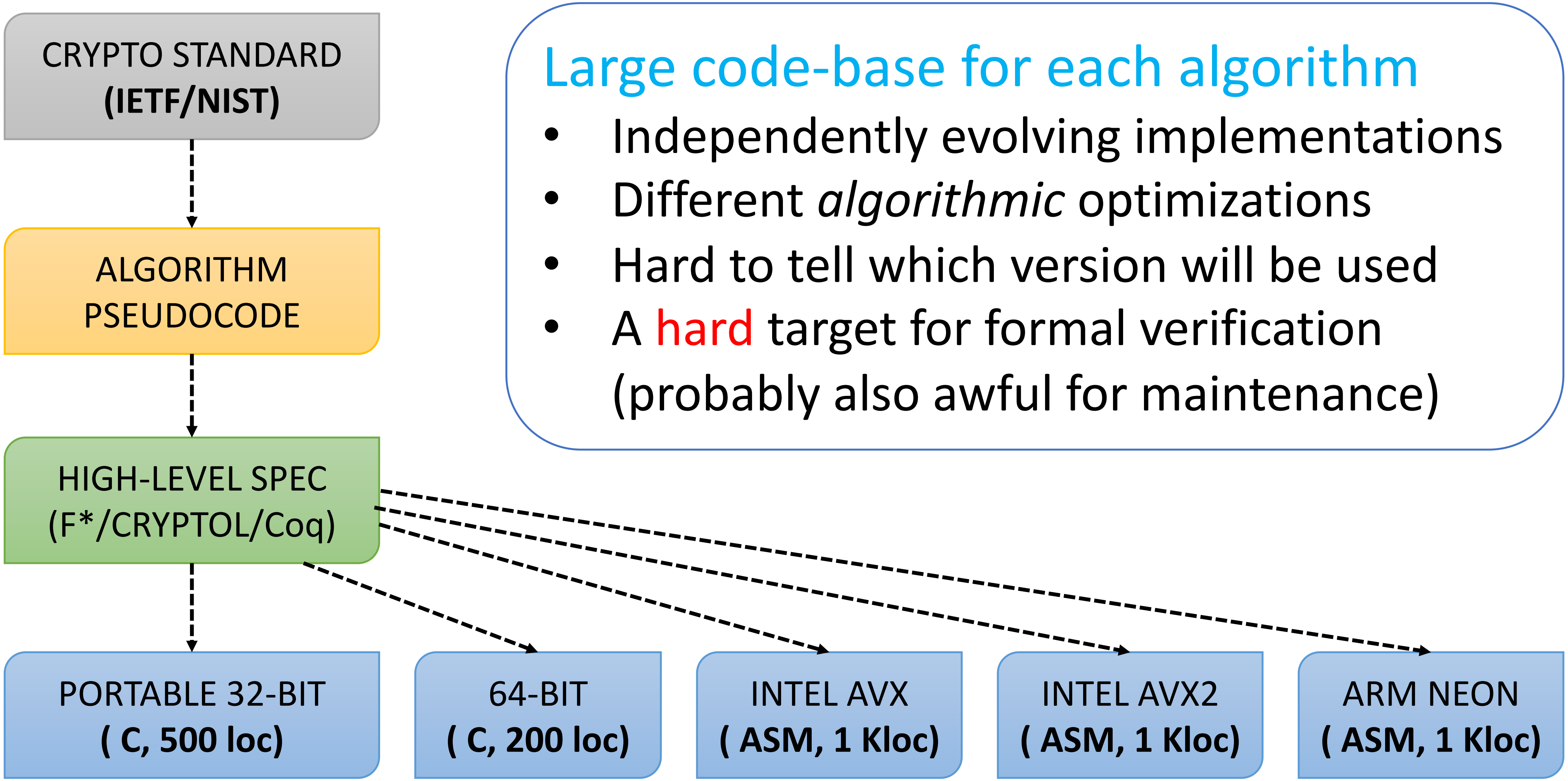
Every line of verified C requires 2x-7x lines of proof

Complex mathematical reasoning interleaved with many boring steps

Many Platform-Specific Implementations

Large code-base for each algorithm

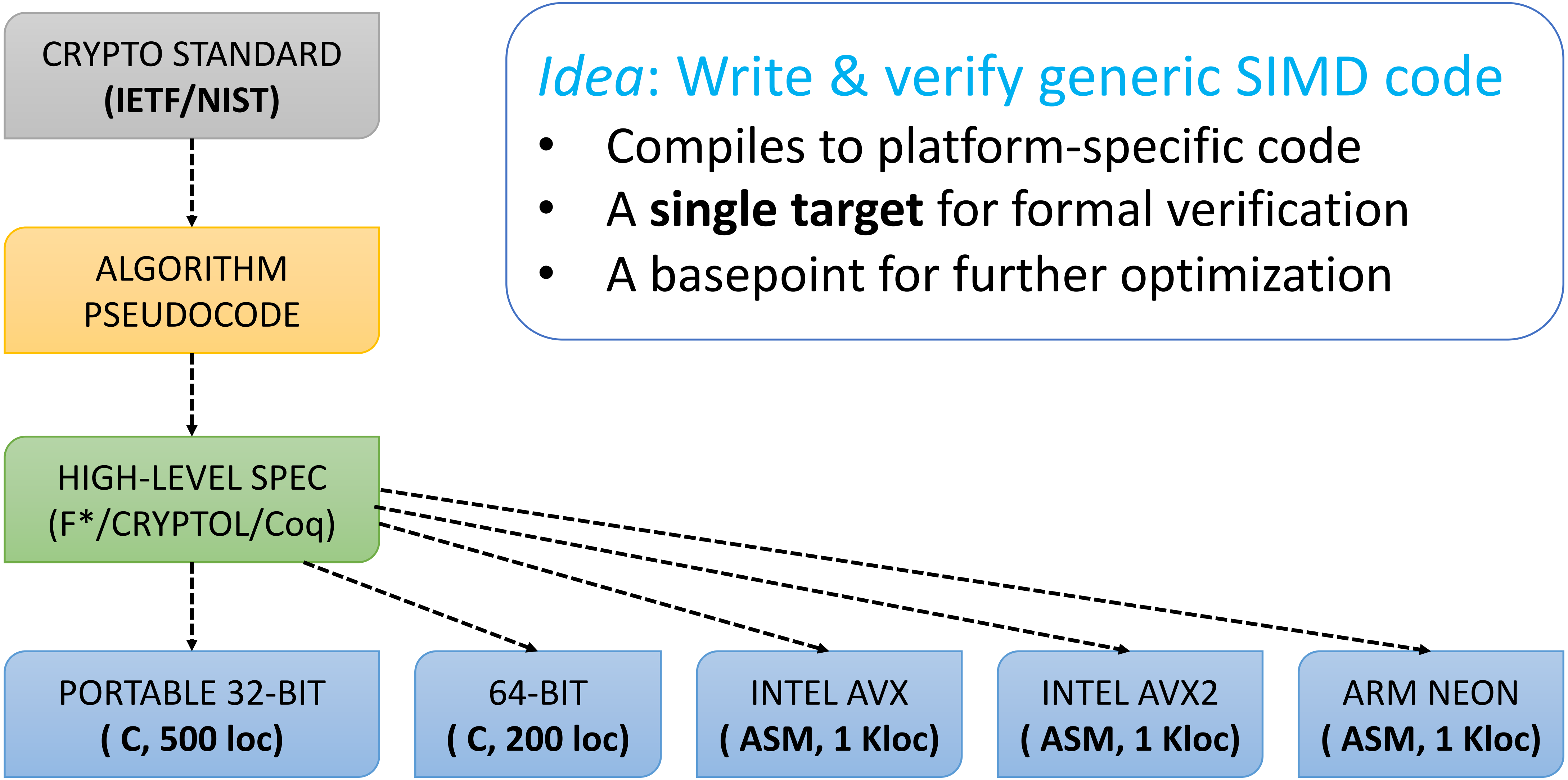
- Independently evolving implementations
- Different *algorithmic* optimizations
- Hard to tell which version will be used
- A hard target for formal verification (probably also awful for maintenance)



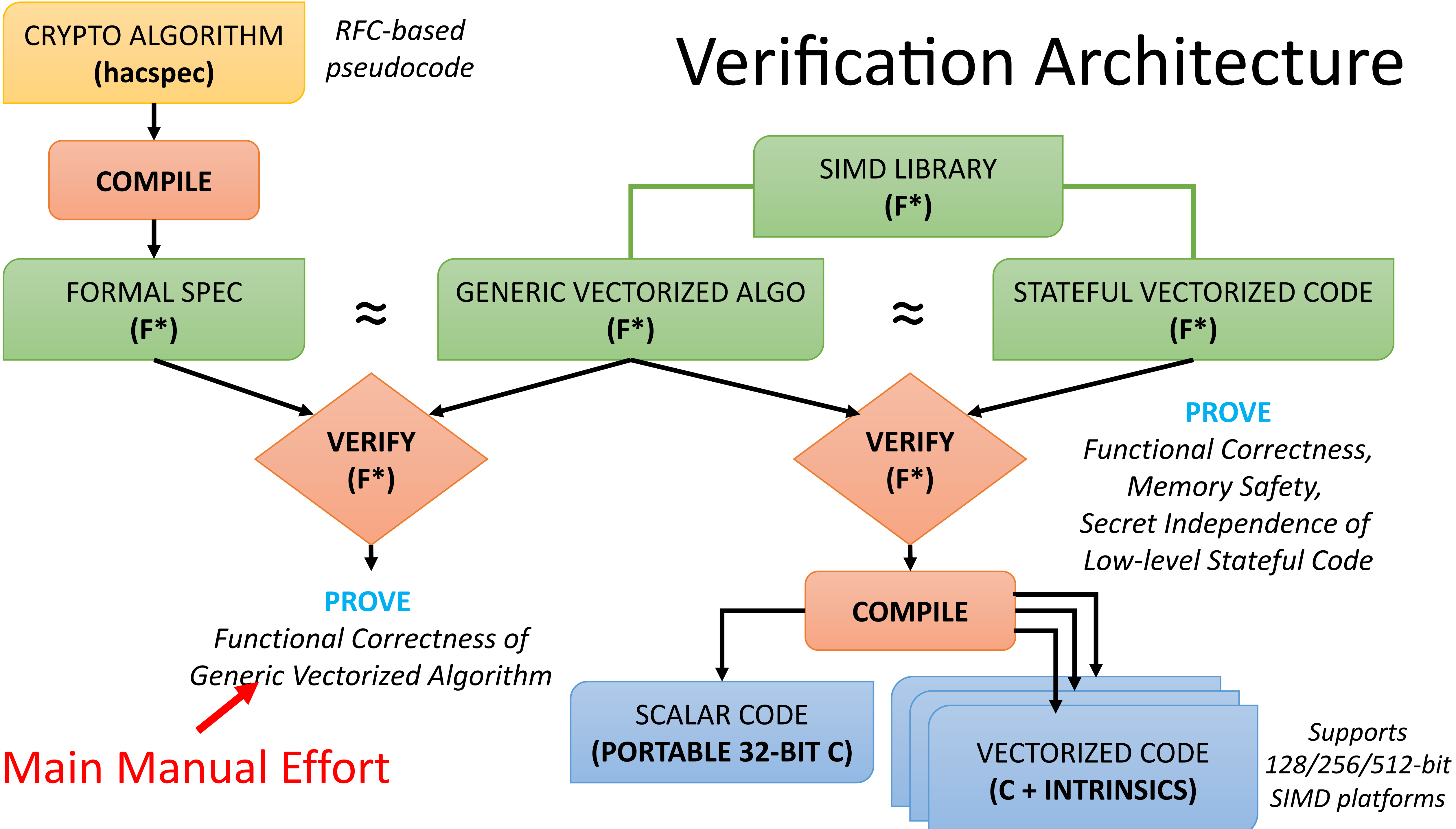
Many Platform-Specific Implementations

Idea: Write & verify generic SIMD code

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



Verification Architecture



F*: a verification oriented language

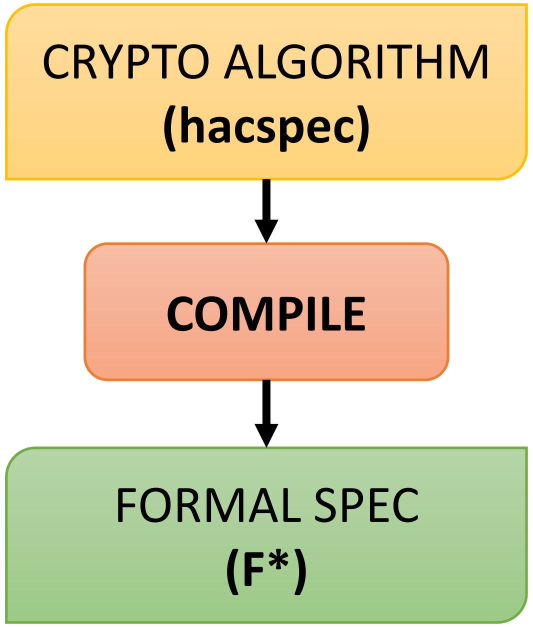
- Functional programming language (« à la Ocaml »)
- Customizable verification system (« à la Coq »)
- Proof automation via SMT solvers (Z3)
- Compilers to Ocaml, F#, C, WebAssembly

http://fstar-lang.org

Actively developed at Microsoft Research and Inria

hacspec: towards verifiable crypto standards

[Bhargavan et al. SSR 2018]



A domain-specific language for writing executable, checkable, formal crypto specs

- Syntactically, a typed subset of Python3
- Looks like the pseudocode used in RFCs

Can be compiled to multiple formal languages

- *Currently:* F* & EasyCrypt. *Next:* Cryptol & Coq
- Allows comparison/composition of different proofs

Add your own spec:

https://github.com/HACS-workshop/hacspec/

Example: CHACHA20 in hacspec

CRYPTO ALGORITHM **(hacspec)**

↓

COMPILE

↓

FORMAL SPEC **(F*)**

```
index_t  = range_t(0,16)
rotval_t = range_t(1,32)
state_t  = array_t(uint32_t,16)

@typechecked
def line(a: index_t, b: index_t, d: index_t,
         s: rotval_t, m: state_t) -> state_t:
    m    = array.copy(m)
    m[a] = m[a] + m[b]
    m[d] = m[d] ^ m[a]
    m[d] = uintn.rotate_left(m[d],s)
    return m
```

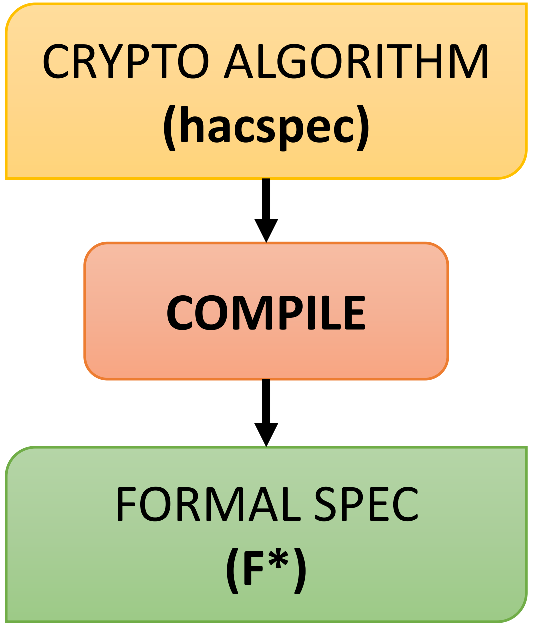
Type Abbreviations

Variable Declarations

Only pure functions: no external side-effects

Code runs in standard python3 with runtime typechecking

Compiled F* spec for CHACHA20




```
let index_t: Type0 = range_t 0 16
let rotval_t: Type0 = range_t 1 32
let state_t: Type0 = array_t uint32_t 16

let line (a: index_t) (b: index_t) (d: index_t)
         (s: rotval_t) (m: state_t) : state_t =
  let m = array_copy m in
  let m = m.[ a ] ← m.[ a ] +. m.[ b ] in
  let m = m.[ d ] ← m.[ d ] ^. m.[ a ] in
  let m = m.[ d ] ← uintn_rotate_left m.[ d ] s in
  m
```

Compiled specification in F* syntax

Types, array bounds, termination statically verified

Vectorization Strategies for CHACHA20

1. Line-level Parallelism
   reorder computations to compute 4 lines in parallel

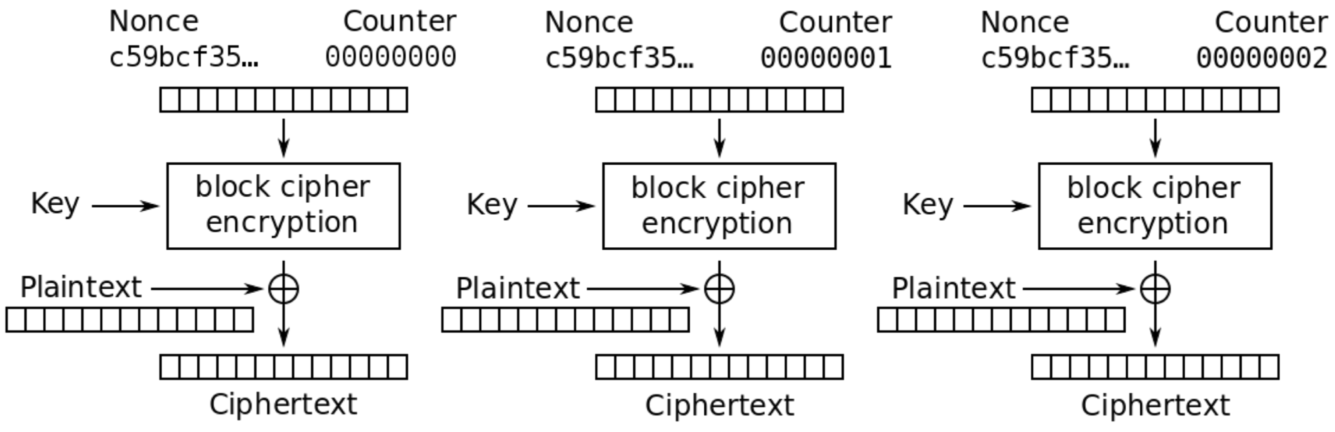
```
let quarter_round (a: index_t) (b: index_t) (c: index_t)
                  (d: index_t) (m: state_t) : state_t =
  let m = line a b d 16 m in
  let m = line c d b 12 m in
  let m = line a b d 8 m in
  let m = line c d b 7 m in
  m

let column_round (m: state_t) : state_t =
  let m = quarter_round 0 4 8 12 m in
  let m = quarter_round 1 5 9 13 m in
  let m = quarter_round 2 6 10 14 m in
  let m = quarter_round 3 7 11 15 m in
  m
```

Vectorization Strategies for CHACHA20

1. Line-level Parallelism
   reorder computations to compute 4 lines in parallel

2. Counter-mode Parallelism
   process any number of blocks in parallel

We implemented both, but 2 is faster and more generic



Counter (CTR) mode encryption

https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation

A Generic Vectorized Algorithm

```
let lanes : Type0 = n:width{n == 1 ∨ n == 4 ∨ n == 8}
let uint32xN (w:lanes) : Type0 =
let state (w:lanes) : Type0 = lseq

let line (#w:lanes) (a:index_t) (b:index_t) (d:index_t)
                    (s:rotval_t) (m:state w) : state w =
  let m = array.copy m in
  let m = m.[ a ] ← m.[ a ] +| m.[ b ] in
  let m = m.[ d ] ← m.[ d ] ^| m.[ a ] in
  let m = m.[ d ] ← uint32xN_rotate_left m.[ d ] s in
  m
```

SUPPORTED VECTOR SIZES

VECTORIZED SPEC

A Generic Vectorized Algorithm

```
let lanes : Type0 = n:width{n == 1 ∨ n == 4 ∨ n == 8}
let uint32xN (w:lanes) : Type0 = vec_t U32 w
```

VECTOR OF w UINT32s

```
pe0 = lseq (uint32xN w) 16

let line (#w:lanes) (a:index_t) (b:index_t) (d:index_t)
                    (s:rotval_t) (m:state w) : state w =
  let m = array.copy m in
  let m = m.[ a ] ← m.[ a ] +| m.[ b ] in
  let m = m.[ d ] ← m.[ d ] ^| m.[ a ] in
  let m = m.[ d ] ← uint32xN_rotate_left m.[ d ] s in
  m
```

VECTORIZED SPEC

A Generic Vectorized Algorithm

```
let lanes : Type0 = n:width{n == 1 ∨ n == 4 ∨ n == 8}
let uint32xN (w:lanes) : Type0 = vec_t U32 w
let state (w:lanes) : Type0 = lseq (uint32xN w) 16
                                     (b:index_t) (d:index_t)
                        (s:rotval_t) (m:state w) : state w =
  let m = array.copy m in
  let m = m.[ a ] ← m.[ a ] +| m.[ b ] in
  let m = m.[ d ] ← m.[ d ] ^| m.[ a ] in
  let m = m.[ d ] ← uint32xN_rotate_left m.[ d ] s in
  m
```

CONTAINS w CHACHA20 STATES

VECTORIZED SPEC

A Generic Vectorized Algorithm

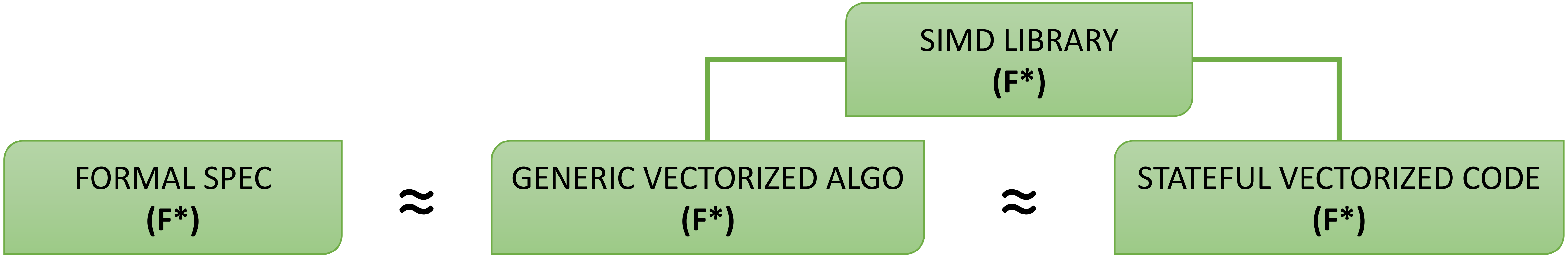
```
let lanes : Type0 = n:width{n == 1 ∨ n == 4 ∨ n == 8}
let uint32xN (w:lanes) : Type0 = vec_t U32 w
let state (w:lanes) : Type0 = lseq (uint32xN w) 16

let line (#w:lanes) (a:index_t) (b:index_t) (d:index_t)
                    (s:rotval_t) (m:state w) : state w =
  let m = array.copy m in
  let m = m.[ a ] ← m.[ a ] +| m.[ b ] in
  let m = m.[ d ] ← m.[ d ] ... [ a ] in
  let m = m.[ d ] ← ...
  m
```

SIMD OP: APPLY TO EACH VECTOR ELEMENT

VECTORIZED SPEC

A Generic Vectorized Algorithm

1. Define SIMD versions of all core functions
   (relying on generic SIMD operations)
2. Define functions to load and store vectorized state
   (using a generic matrix transposition library)
3. Modify Counter-Mode Encryption to process w blocks at once



Verifying the Vectorized Algorithm

1. Prove lemmas showing that each vectorized function maps over the corresponding scalar function

2. Prove lemmas showing that that the main API functions have the same input-output behavior

SCALAR SPEC
(F*)

≈

GENERIC VECTORIZED ALGO
(F*)

Verifying the Vectorized Algorithm

1. Prove lemmas showing that each vectorized function maps over the corresponding scalar function

```
val line_lemma: #w:lanes → a:index_t → b:index_t → d:index_t →
                             s:rotval_t → m:state w →
    Lemma (transpose_state (line #w a b d s m) ==
           map (Scalar.line a b d s) (transpose_state m))
```

SCALAR SPEC
(F*)

≈

GENERIC VECTORIZED ALGO
(F*)

Verifying the Vectorized Algorithm

```
val chacha20_encrypt_bytes_lemma: #w:lanes →
    k:key → n:nonce → c:counter →
    msg:bytes{length msg/size_block ≤ max_size_t} →
    Lemma (chacha20_encrypt_bytes #w k n c msg ==
           Scalar.chacha20_encrypt_bytes k n c msg)
```

2. Prove lemmas showing that that the main API functions have the same input-output behavior

SCALAR SPEC **(F*)** ≈ GENERIC VECTORIZED ALGO **(F*)**

From Algorithm to Vectorized Code

```
inline_for_extraction
val line: #w:lanes → st:state w →
          a:index → b:index → d:index →
          r:rotval U32 → ST unit
          (requires (λ h → live h st))
          (ensures (λ h0 _ h1 → modifies (loc st) h0 h1 ∧
                        as_seq h1 st ==
                        Spec.line (v a) (v b) (v d)
                                  r (as_seq h0 st)))
let line #w st a b d r =
  st.(a) ← st.(a) +| st.(b);
  st.(a) ← st.(a) ^| st.(d);
  st.(d) ← st.(d) <<<| r
```

From Algorithm to Vectorized Code

```
inline_for_extraction
val line: #w:lanes → st:state w →
          a:index → b:inde
          r:rotval U32 → S
          (requires (λ h → live h st))
          (ensures (λ h0 _ h1 → modifies (loc st) h0 h1 ∧
                   as_seq h
                   Spec.line
                             r (as_seq h0 st)))

let line #w st a b d r =
  st.(a) ← st.(a) +| st.(b);
  st.(a) ← st.(a) ^| st.(d);
  st.(d) ← st.(d) <<<| r
```

MEMORY SAFETY PRECONDITION

MEMORY SAFETY POSTCONDITION

From Algorithm to Vectorized Code

```
inline_for_extraction
val line: #w:lanes → st:state w →
          a:index → b:index → d:index →
          r:rotval U32 → ST unit
          (requires (λ h → live h st))
          (ensures (λ h0 _ h1 → modifies (loc st) h0 h1 ∧
                   as_seq h1 st ==
                   Spec.line (v a) (v b) (v d)
                             r (as_seq h0 st)))
let line #w st a b d r =
  st.(a) ← st.(a) +| st.(b);
  st.(a) ← st.(a) ^| st.(d);
  st
```

FUNCTIONAL CORRECTNESS GOAL

F* VERIFIES THAT GENERIC STATEFUL CODE MEETS ITS SPEC

Generating C Code for Different Platforms

```c
inline static void Hacl_Impl_Chacha20_Core32xN_double_round1(uint32_t *st)
{
  uint32_t sta0 = st[0U];
  uint32_t stb0 = st[4U];
  uint32_t std0 = st[12U];
  uint32_t sta10 = sta0 + stb0;
  uint32_t std10 = std0 ^ sta10;
  uint32_t std20 = std10 << (uint32_t)16U | std10 >> ((uint32_t)32U - (uint32_t)16U);
```

w = 1: 32-BIT SCALAR CODE IN PORTABLE C

```c
inline static void
Hacl_Impl_Chacha20_Core32xN_double_round4(Lib_IntVector_Intrinsics_vec128 *st)
{
  Lib_IntVector_Intrinsics_vec128 sta0 = st[0U];
  Lib_IntVector_Intrinsics_vec128 stb0 = st[4U];
  Lib_IntVector_Intrinsics_vec128 std0 = st[12U];
  Lib_IntVector_Intrinsics_vec128 sta10 = Lib_IntVector_Intrinsics_vec128_add32(sta0, stb0);
  Lib_IntVector_Intrinsics_vec128 std10 = Lib_IntVector_Intrinsics_vec128_xor(std0, sta10);
  Lib_IntVector_Intrinsics_vec128
  std20 =
    Lib_IntVector_Intrinsics_vec128_or(Lib_IntVector_Intrinsics_vec128_shift_left32(std10,
        (uint32_t)16U),
      Lib_IntVector_Intrinsics_vec128_shift_right32(std10, (uint32_t)32U - (uint32_t)16U));
```

w = 4: 128-BIT VECTORIZED CODE USING AVX/NEON INTRINSICS

Generating C Code for Different Platforms

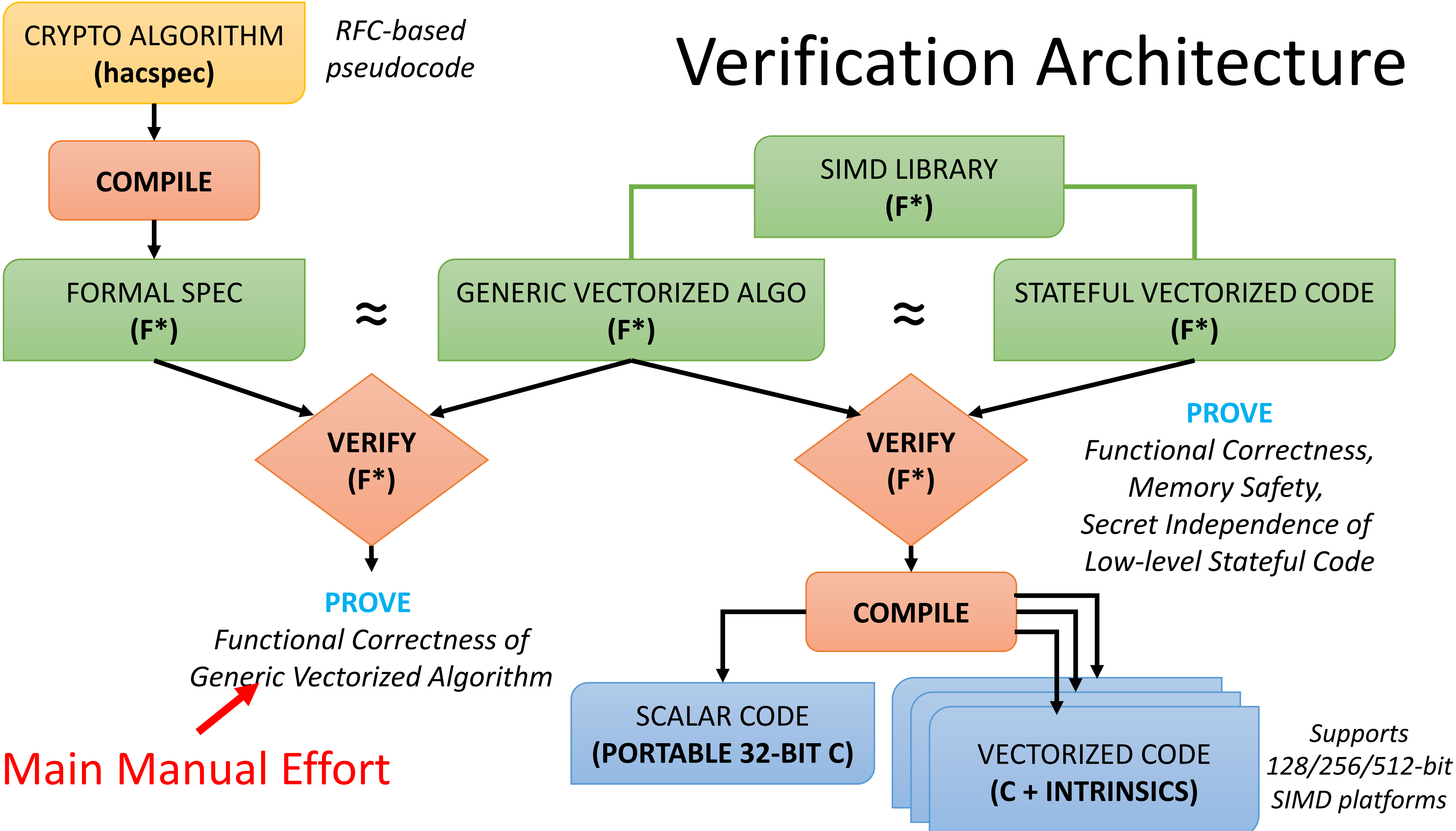
```
inline static void Hacl_Impl_Chacha20_Core32xN_double_round1(uint32_t *st)
{
  uint32_t sta0 = st[0U];
  uint32_t stb0 = st[4U];
  uint32_t std0 = st[12U];
  uint32_t sta10 = sta0 + stb0;
  uint32_t std10 = std0 ^ sta10;
  uint32_t std20 = std10 << (uint32_t)16U | std10 >> ((uint32_t)32U - (uint32_t)16U);
```

w = 1: 32-BIT SCALAR CODE IN PORTABLE C

```
inline static void
Hacl_Impl_Chacha20_Core32xN_double_round8(Lib_IntVector_Intrinsics_vec256 *st)
{
  Lib_IntVector_Intrinsics_vec256 sta0 = st[0U];
  Lib_IntVector_Intrinsics_vec256 stb0 = st[4U];
  Lib_IntVector_Intrinsics_vec256 std0 = st[12U];
  Lib_IntVector_Intrinsics_vec256 sta10 = Lib_IntVector_Intrinsics_vec256_add32(sta0, stb0);
  Lib_IntVector_Intrinsics_vec256 std10 = Lib_IntVector_Intrinsics_vec256_xor(std0, sta10);
  Lib_IntVector_Intrinsics_vec256
  std20 =
    Lib_IntVector_Intrinsics_vec256_or(Lib_IntVector_Intrinsics_vec256_shift_left32(std10,
        (uint32_t)16U),
      Lib_IntVector_Intrinsics_vec256_shift_right32(std10, (uint32_t)32U - (uint32_t)16U));
```

w = 8: 256-BIT VECTORIZED CODE USING AVX2 INTRINSICS

Verification Architecture



Verifying Vectorized POLY1305

1. Verify vectorized field arithmetic

   Each function calculates w field operations in parallel

2. Exploit inherent parallelism in polynomial evaluation

   Transform the poly1305 loop using Horner's rule (1x/2x/4x)

3. Prove that the vectorized MAC returns the correct value

SCALAR SPEC **(F*)** ≈ GENERIC VECTORIZED SPEC **(F*)**

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 1 scalar implementation

Ongoing Work

We are systematically applying our new approach to write generic vectorized code for most of HACL*

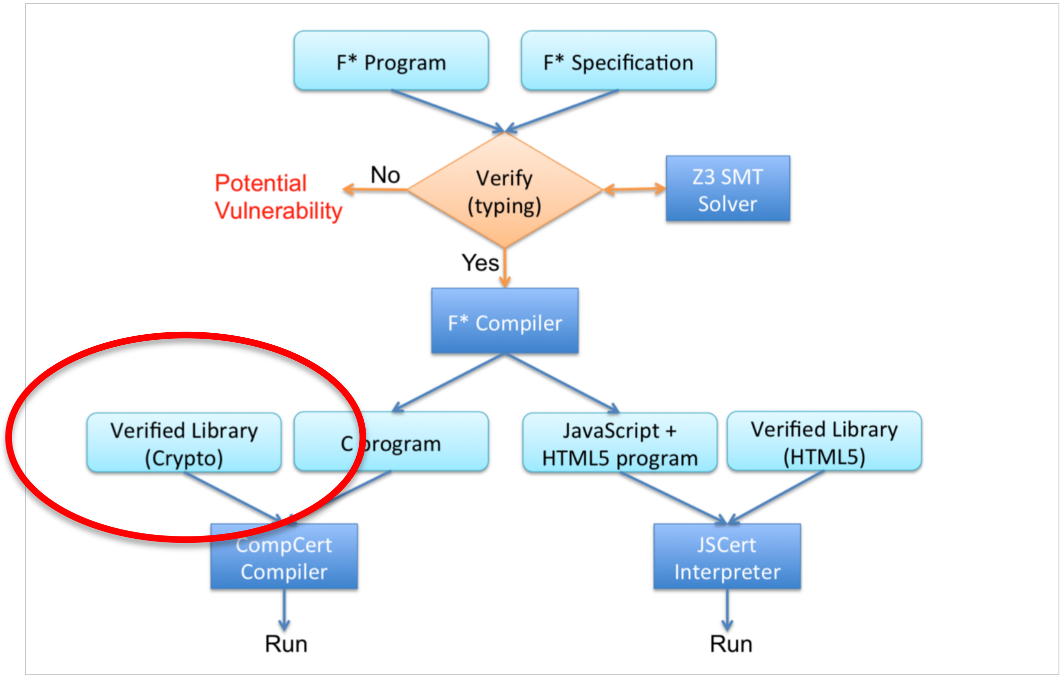
- New implementations of AES-GCM, SHA-2, SHA-3, …
- Ongoing deployments to Firefox, WireGuard, Fizz, …

Verified crypto feeds into larger verification projects

- New verified constructions: Post-Quantum Crypto
- New verified protocols: Signal, TLS 1.3, Noise
- New target platforms: WebAssembly

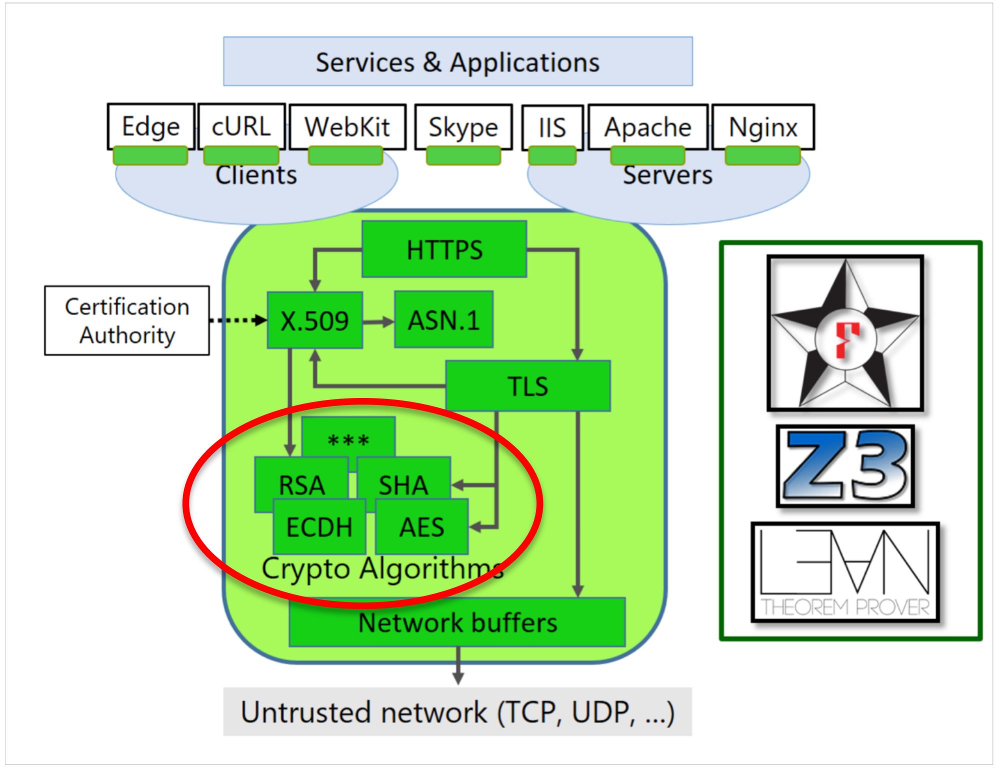
ERC CIRCUS [2016-21]

Building Verified Cryptographic Web Applications



Project Everest [2016-20]

Building a Verified HTTPS Stack



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

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

If you are a crypto developer:
consider writing generic optimized algorithms

- Don't just dump more unverified assembly into the library

Questions?

- HACL*: https://github.com/project-everest/hacl-star
- hacspec: https://github.com/HACS-workshop/hacspec
- F*: https://www.fstar-lang.org

- INRIA PROSECCO: http://prosecco.inria.fr
- Microsoft Project Everest: https://project-everest.github.io/