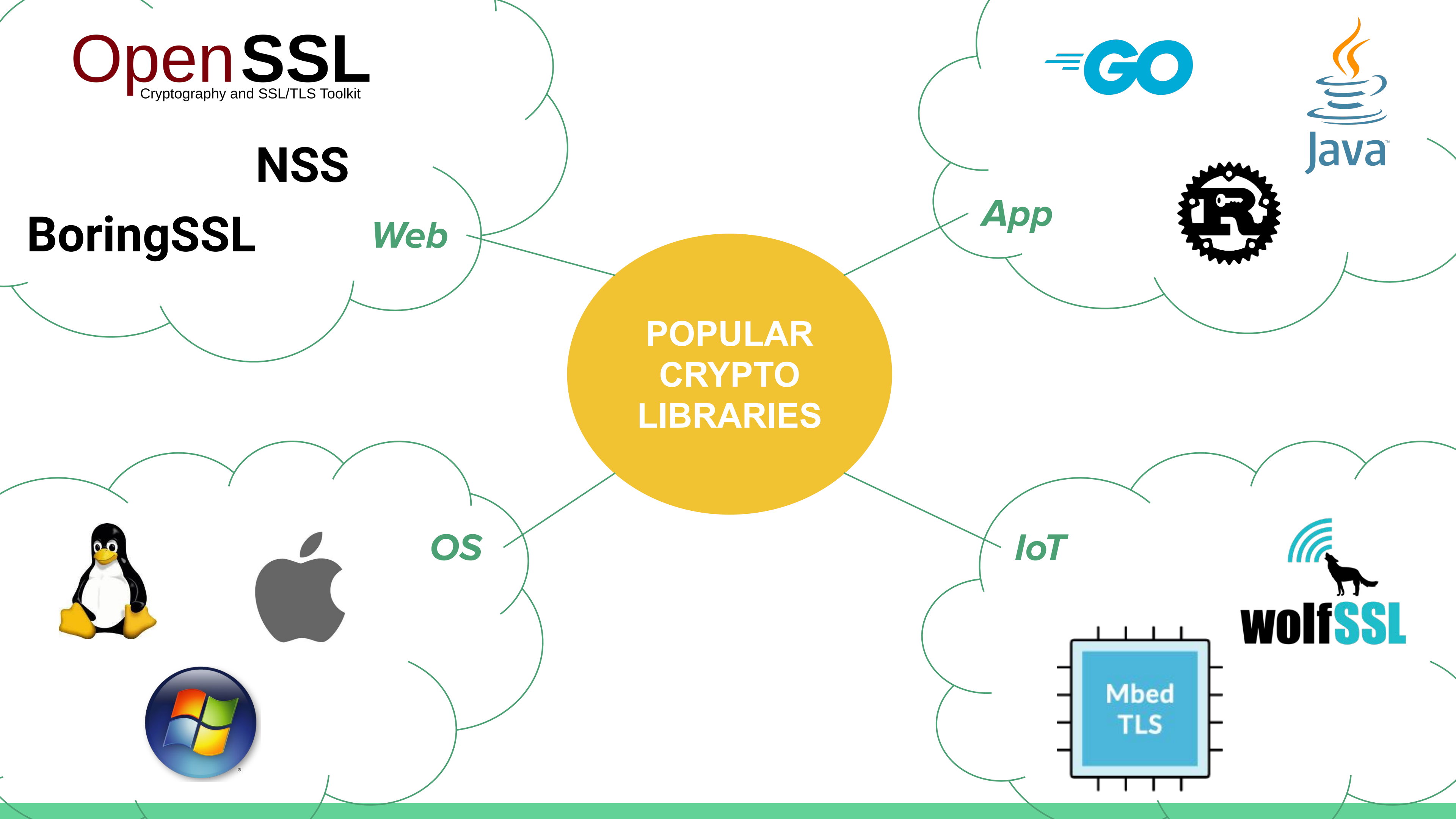
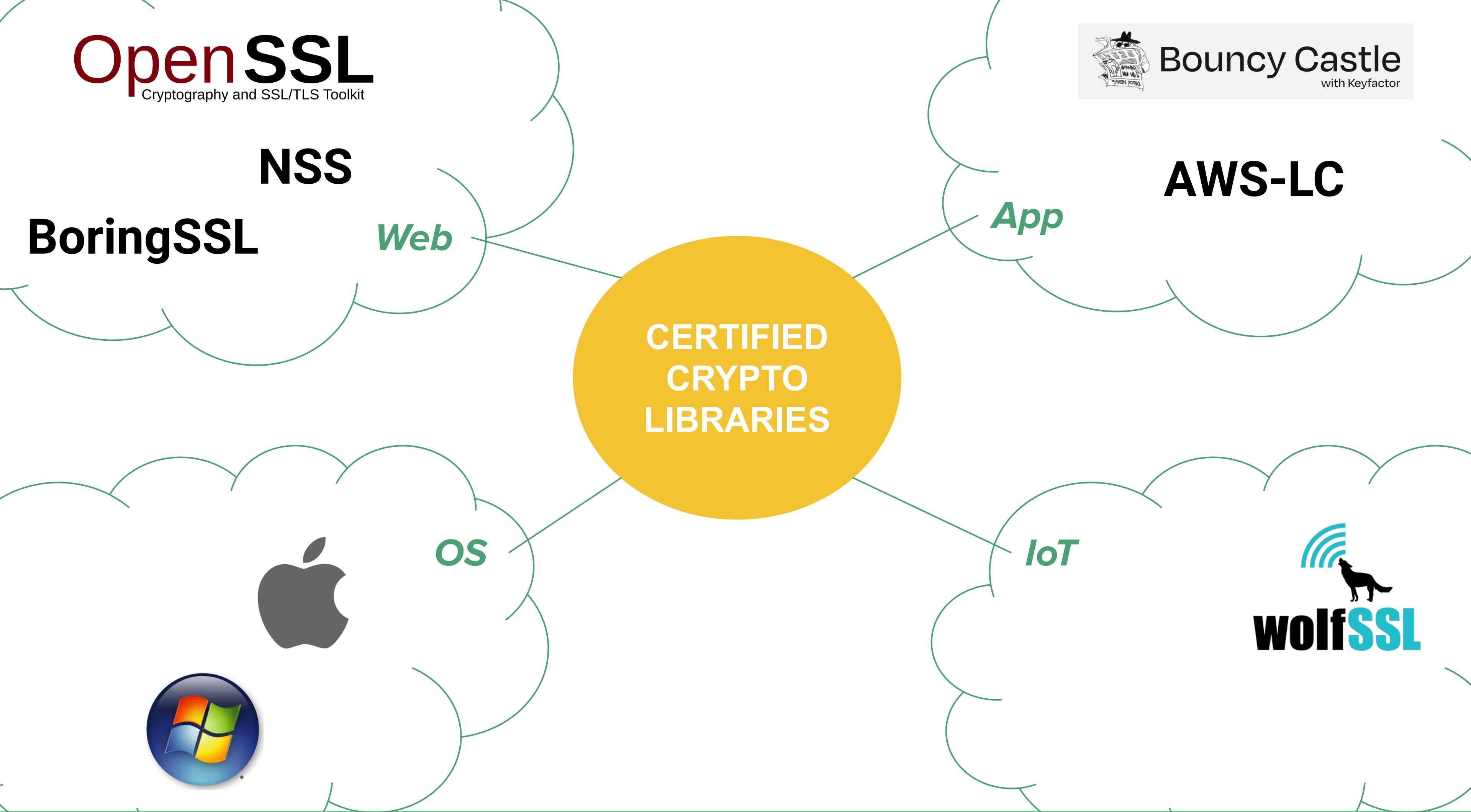
Formal Specifications for Certifiable Cryptography

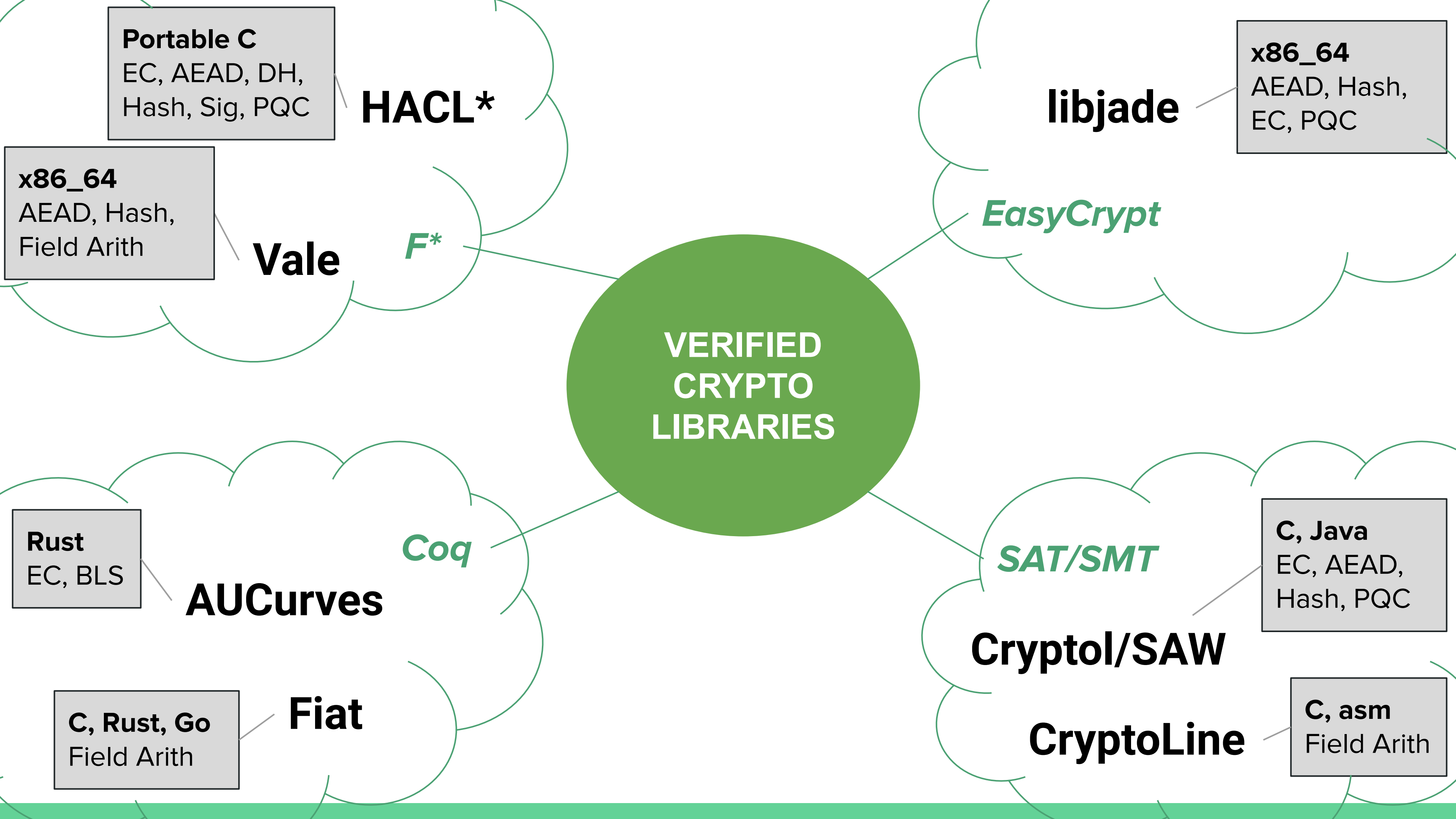
Karthikeyan Bhargavan

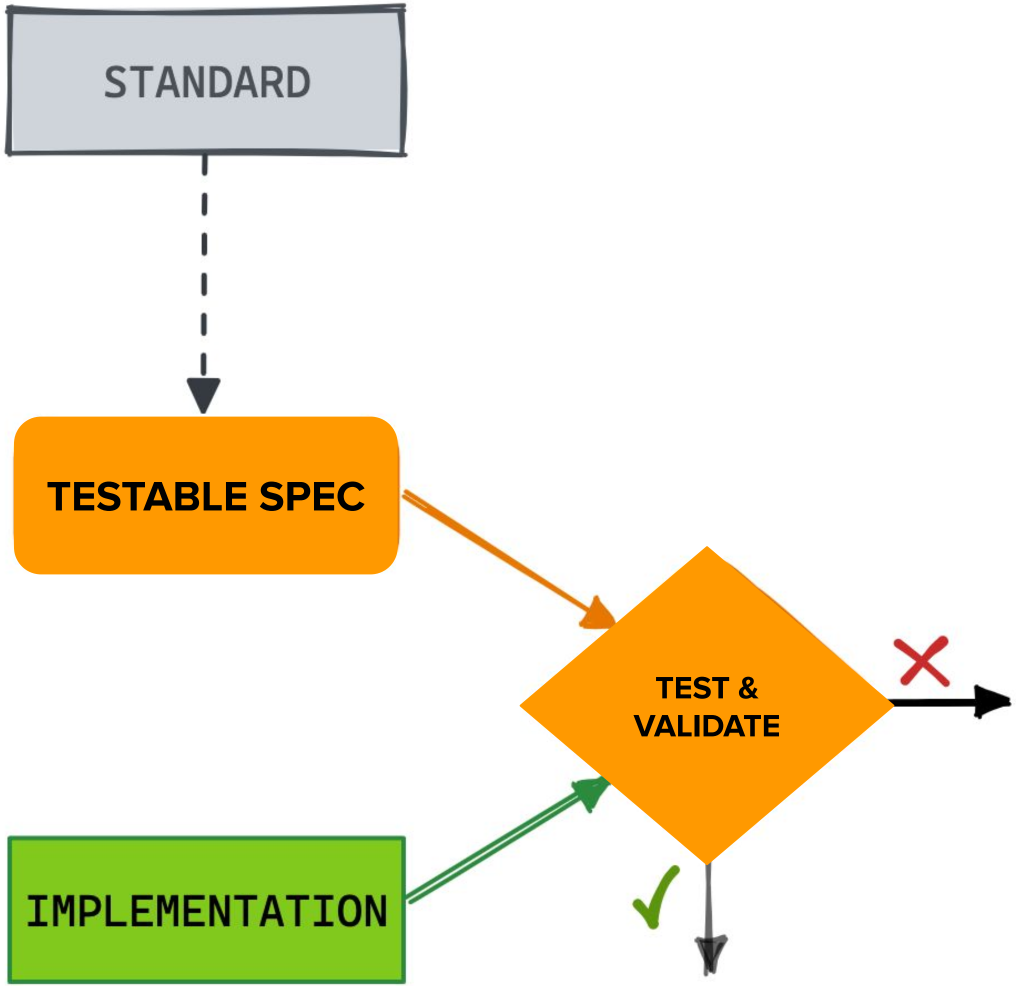
Manuel Barbosa, Franziskus Kiefer,
Peter Schwabe, Pierre-Yves Strub


POPULAR CRYPTO LIBRARIES
Web
OpenSSL
Cryptography and SSL/TLS Toolkit
NSS
BoringSSL
App
GO
Java
OS
IoT
wolfSSL
Mbed TLS



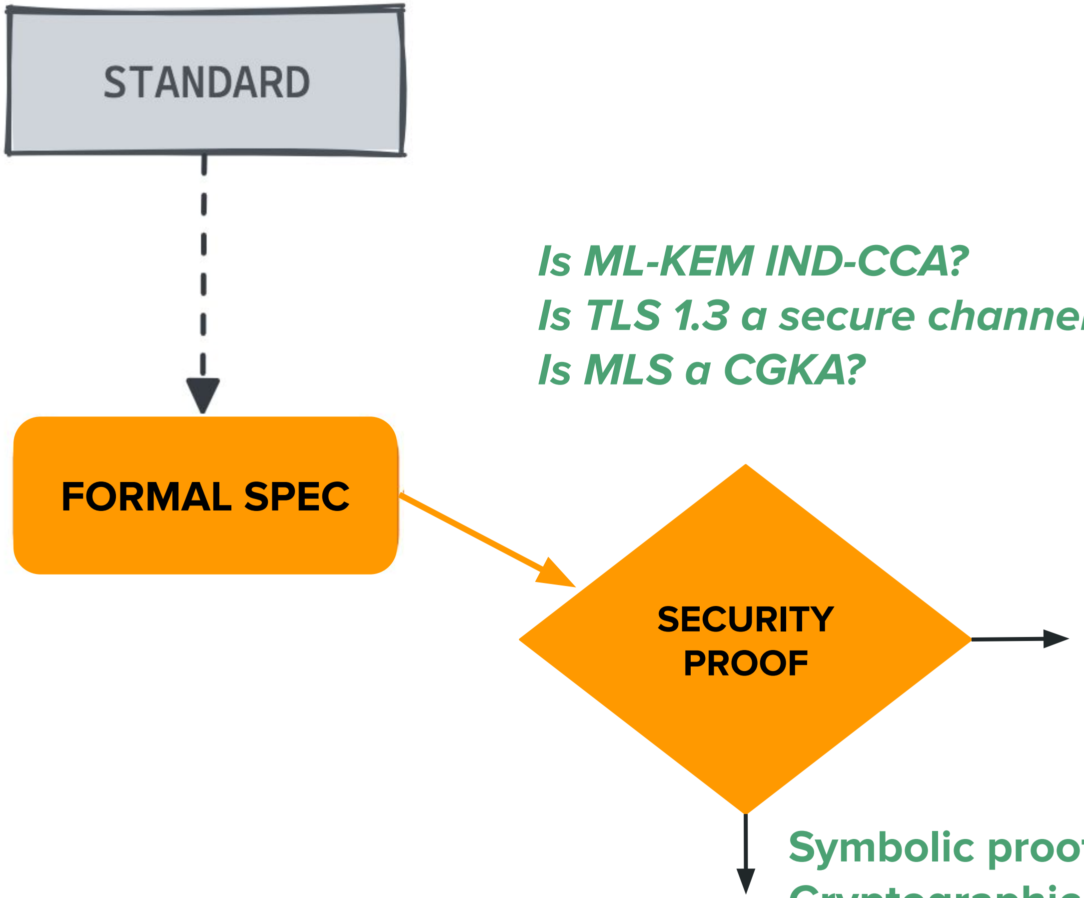
OpenSSL
Cryptography and SSL/TLS Toolkit
NSS
BoringSSL
Web
Bouncy Castle
with Keyfactor
AWS-LC
App
CERTIFIED CRYPTO LIBRARIES
OS
IoT
wolfSSL


VERIFIED CRYPTO LIBRARIES
HACL*
Portable C
EC, AEAD, DH, Hash, Sig, PQC
Vale
x86_64
AEAD, Hash, Field Arith
F*
libjade
x86_64
AEAD, Hash, EC, PQC
EasyCrypt
Coq
AUCurves
Rust
EC, BLS
Fiat
C, Rust, Go
Field Arith
SAT/SMT
Cryptol/SAW
C, Java
EC, AEAD, Hash, PQC
CryptoLine
C, asm
Field Arith



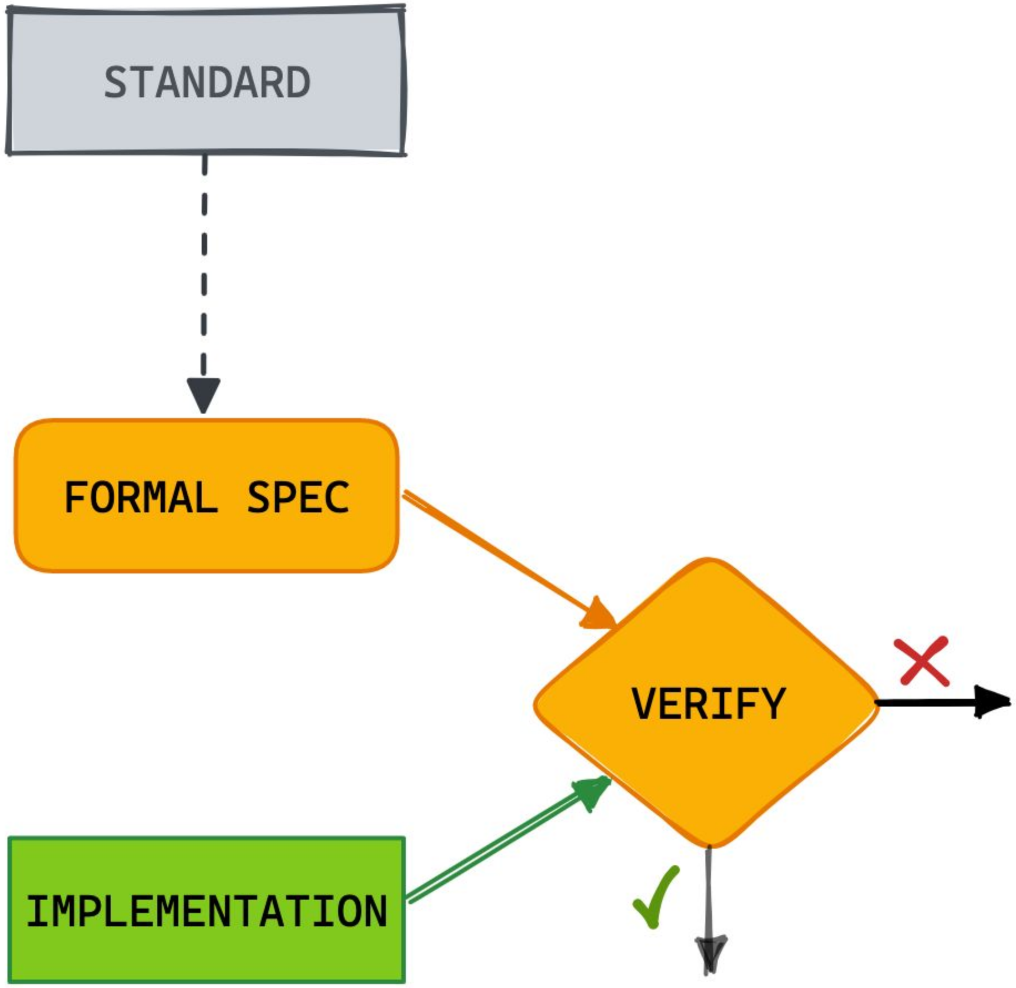
Certification Workflow

Security Analysis Workflow

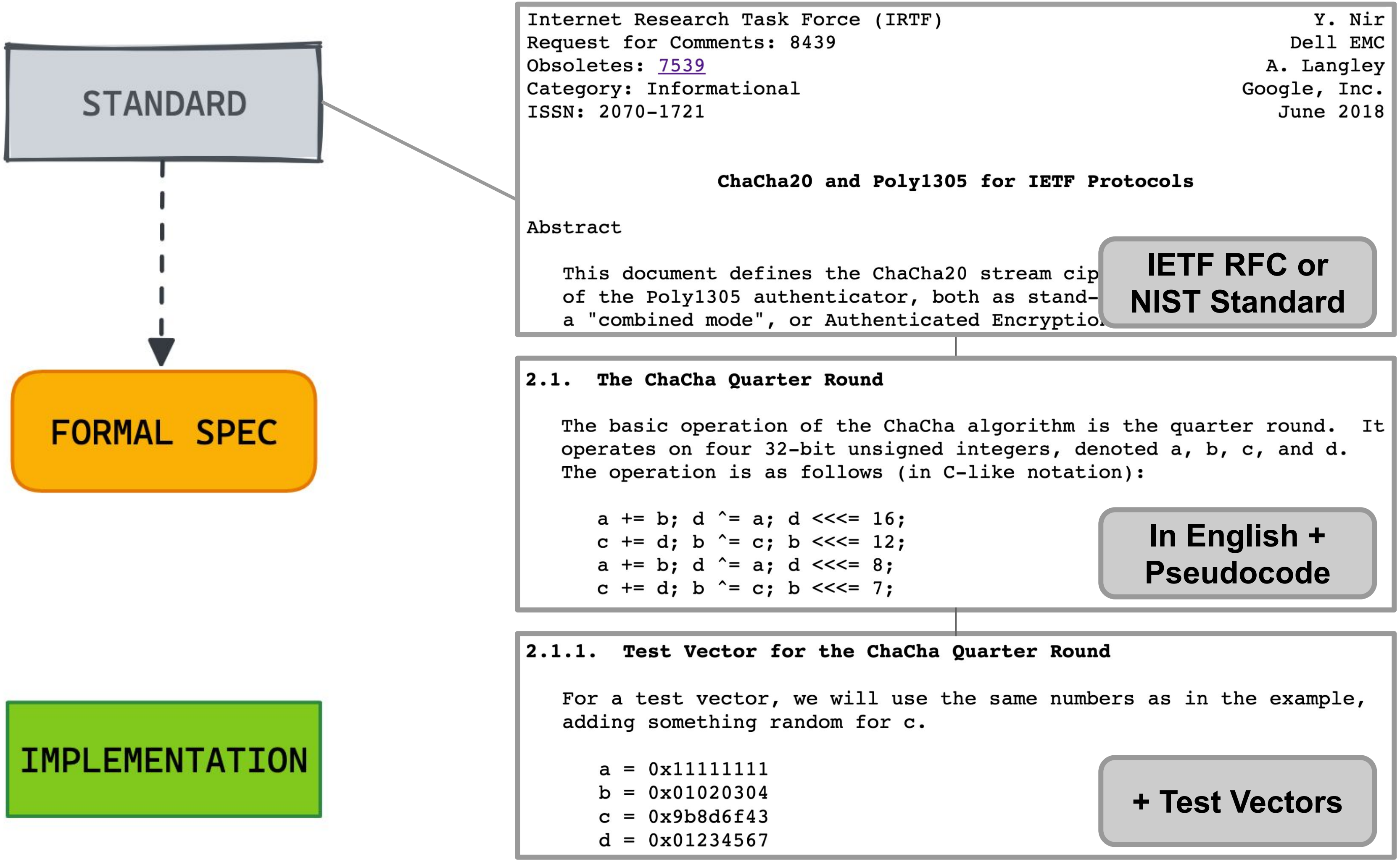


Is ML-KEM IND-CCA?
Is TLS 1.3 a secure channel?
Is MLS a CGKA?

Symbolic proofs: ProVerif/Tamarin
Cryptographic proofs: EasyCrypt/CryptoVerif/Squirrel
Post-Quantum proofs: EasyCrypt/CryptoVerif/Squirrel



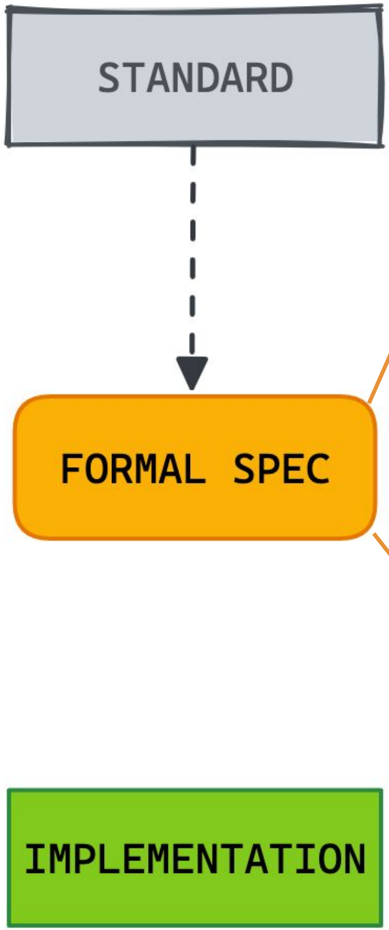
Verified Cryptography Workflow




```
Internet Research Task Force (IRTF)                          Y. Nir
Request for Comments: 8439                                 Dell EMC
Obsoletes: 7539                                          A. Langley
Category: Informational                                 Google, Inc.
ISSN: 2070-1721                                           June 2018


              ChaCha20 and Poly1305 for IETF Protocols

Abstract

   This document defines the ChaCha20 stream cip
   of the Poly1305 authenticator, both as stand-
   a "combined mode", or Authenticated Encryptio
```

```
2.1.  The ChaCha Quarter Round

   The basic operation of the ChaCha algorithm is the quarter round.  It
   operates on four 32-bit unsigned integers, denoted a, b, c, and d.
   The operation is as follows (in C-like notation):

      a += b; d ^= a; d <<<= 16;
      c += d; b ^= c; b <<<= 12;
      a += b; d ^= a; d <<<= 8;
      c += d; b ^= c; b <<<= 7;
```

```
2.1.1.  Test Vector for the ChaCha Quarter Round

   For a test vector, we will use the same numbers as in the example,
   adding something random for c.

      a = 0x11111111
      b = 0x01020304
      c = 0x9b8d6f43
      d = 0x01234567
```

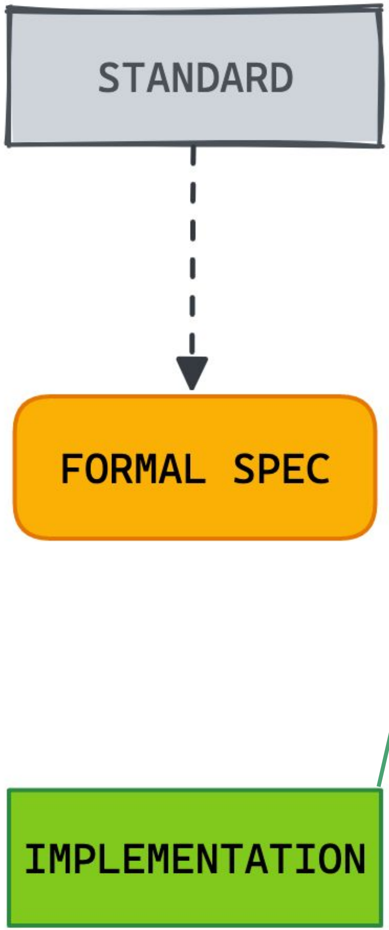



```
let line (a:idx) (b:idx) (d:idx) (s:rotval U32) (m:state) : Tot state =
  let m = m.[a] ← (m.[a] +. m.[b]) in
  let m = m.[d] ← ((m.[d] ^. m.[a]) <<<. s) in m

let quarter_round a b c d : Tot shuffle =
  line a b d (size 16) @
  line c d b (size 12) @
  line a b d (size 8)  @
  line c d b (size 7)
```

F* Spec
(HACL*)

```
proc chacha20_line(a : int, b : int, d : int, s : int, st : State) = {
  var state;
  state <- st;
  state.[a] <- ((state).[a]) + ((state).[b]);
  state.[d] <- ((state).[d]) `^` ((state).[a]);
  state.[d] <- rotate_left ((state).[d]) (s);
  return state;
}

proc chacha20_quarter_round(a : int, b : int, c : int, d : int, st : State) = {
  var state;
  state <@ chacha20_line (a, b, d, 16, st);
  state <@ chacha20_line (c, d, b, 12, state);
  state <@ chacha20_line (a, b, d, 8, state);
  state <@ chacha20_line (c, d, b, 7, state);
  return state;
}
```

EasyCrypt Spec
(libjade)



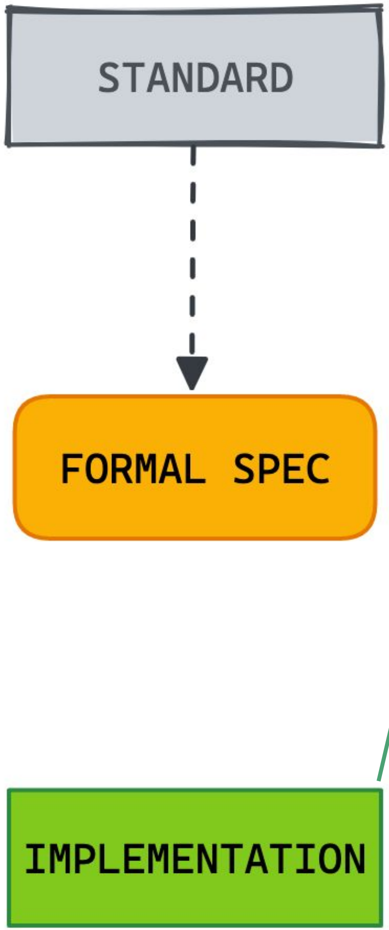
F* Implementation

```
let line st a b d r =
  let sta = st.(a) in
  let stb = st.(b) in
  let std = st.(d) in
  let sta = sta +. stb in
  let std = std ^. sta in
  let std = rotate_left std r in
  st.(a) ← sta;
  st.(d) ← std

let quarter_round st a b c d =
  line st a b d (size 16);
  line st c d b (size 12);
  line st a b d (size 8);
  line st c d b (size 7)
```

Translate

```c
static inline void quarter_round(uint32_t *st, uint32_t a, uint32_t b, uint32_t c, uint32_t d)
{
  uint32_t sta = st[a];
  uint32_t stb0 = st[b];
  uint32_t std0 = st[d];
  uint32_t sta10 = sta + stb0;
  uint32_t std10 = std0 ^ sta10;
  uint32_t std2 = std10 << (uint32_t)16U | std10 >> (uint32_t)16U;
  st[a] = sta10;
  st[d] = std2;
  ...
```

Portable C Code



Jasmin Implementation

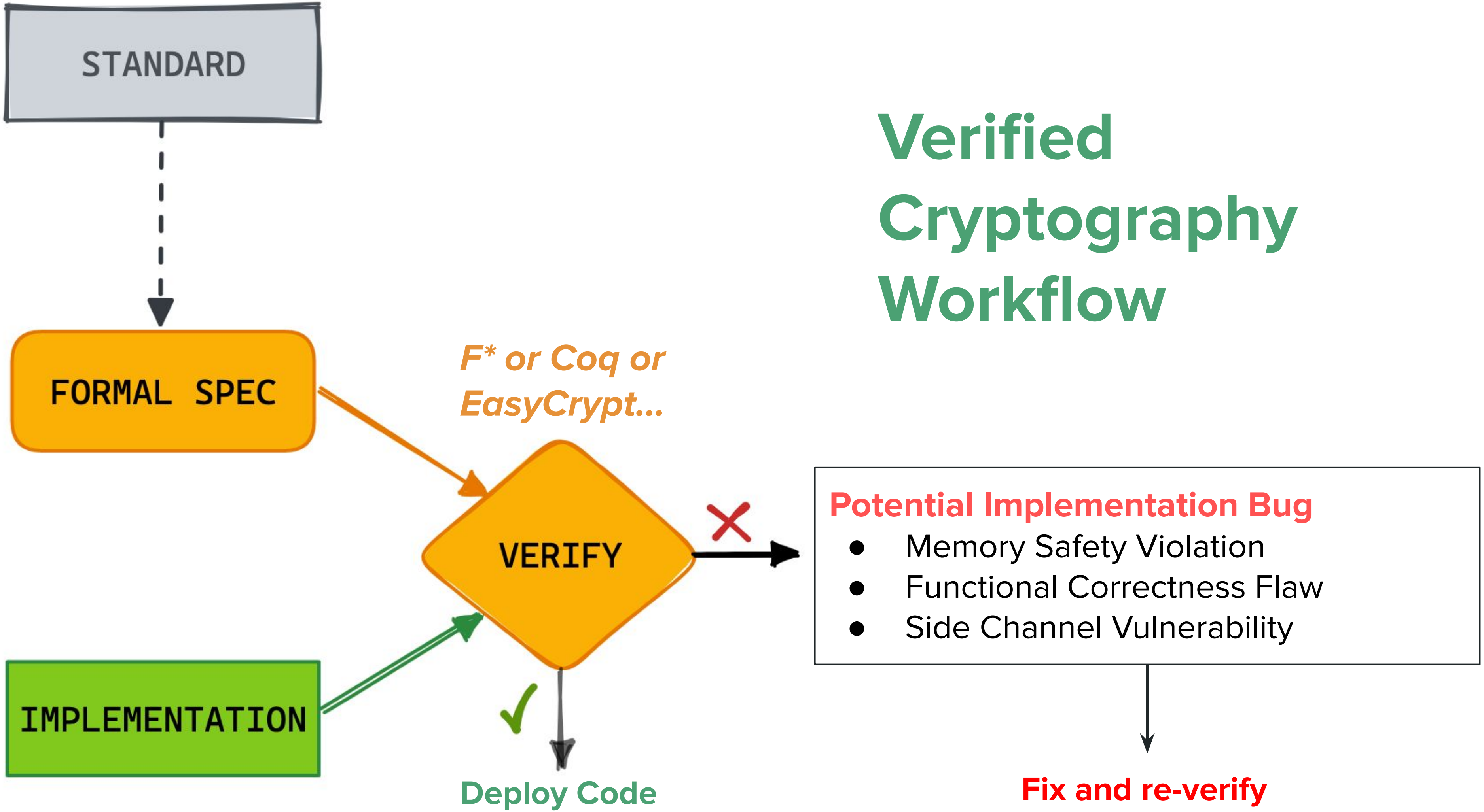
```
inline fn __line_ref(reg u32[16] k,
                     inline int a b c r)
                    -> reg u32[16]
{
  k[a] += k[b];
  k[c] ^= k[a];
  _, _, k[c] = #ROL_32(k[c], r);
  return k;
}

inline fn __quarter_round_ref(reg u32[16] k,
                              inline int a b c d)
                             -> reg u32[16]
{
  k = __line_ref(k, a, b, d, 16);
  k = __line_ref(k, c, d, b, 12);
  k = __line_ref(k, a, b, d, 8);
  k = __line_ref(k, c, d, b, 7);
  return k;
}
```

Translate

Intel AVX2 Assembly

```
vpaddd  %ymm4, %ymm0, %ymm0
vpxor   %ymm0, %ymm12, %ymm12
vpshufb (%rsp), %ymm12, %ymm12
vpaddd  %ymm12, %ymm8, %ymm8
vpaddd  %ymm6, %ymm2, %ymm2
vpxor   %ymm8, %ymm4, %ymm4
vpxor   %ymm2, %ymm14, %ymm14
vpslld  $12, %ymm4, %ymm15
vpsrld  $20, %ymm4, %ymm4
vpxor   %ymm15, %ymm4, %ymm4
vpshufb (%rsp), %ymm14, %ymm14
vpaddd  %ymm4, %ymm0, %ymm0
vpaddd  %ymm14, %ymm10, %ymm10
vpxor   %ymm0, %ymm12, %ymm12
vpxor   %ymm10, %ymm6, %ymm6
vpshufb 32(%rsp), %ymm12, %ymm12
vpslld  $12, %ymm6, %ymm15
vpsrld  $20, %ymm6, %ymm6
...
```

Verified Cryptography Workflow



Good news: For any modern crypto algorithm, there is probably a verified implementation

- You don't have to sacrifice performance
- Mechanized proofs that you can run and re-run yourself
- You (mostly) don't have to read or understand the proofs

HACL* and libcrux

- **HACL*:** Verified C/assembly implementations of all the classical crypto you need
  - Specs/Proofs in F*
  - Intel/ARM SIMD-optimized
  - “Fastest in the world” (sometimes)

- **libcrux:** Verified Rust (and C) implementations of modern FIPS algorithms: SHA-3, ML-KEM, FrodoKEM, …

- Used in Firefox, Linux, etc.

Algorithm	Portable C code	Arm A64 Neon	Intel x64 AVX	Intel x64 AVX2	Intel x64 AVX512	Intel x64 Vale
AEAD						
Chacha20-Poly1305	✓ [43] (+)	✓ (*)	✓ (*)	✓ (*)	✓ (*)	
AES-GCM						✓ [20]
Hashes						
SHA-224,256	✓ [43] (+)	✓ (*)	✓ (*)	✓ (*)	✓ (*)	✓ [20]
SHA-384,512	✓ [43] (+)	✓ (*)	✓ (*)	✓ (*)	✓ (*)	
Blake2s, Blake2b	✓ [34] (+)	✓ (*)	✓ (*)	✓ (*)		
SHA3-224,256,384,512	✓ [34]					
HMAC and HKDF						
HMAC (SHA-2,Blake2)	✓ [43]	✓ (*)	✓ (*)	✓ (*)	✓ (*)	
HKDF (SHA-2,Blake2)	✓ [43]	✓ (*)	✓ (*)	✓ (*)	✓ (*)	
ECC						
Curve25519	✓ [43]					✓ [34]
Ed25519	✓ [43]					
P-256	✓ [34]					
High-level APIs						
Box	✓ [43]					
HPKE	✓ (*)	✓ (*)	✓ (*)	✓ (*)	✓ (*)	✓ (*)

But… not always easy to use, extend, or combine code from verified libraries

- You do need to carefully audit the formal specs, written in tool-specific spec languages like F*, Coq, EasyCrypt
- You do need to safely use their low-level APIs, which often embed subtle security-critical pre-conditions

Specs are needed for analysis and verification

But… what makes a spec a (good) spec?

Specs for ML-KEM

Mathematical Operations

$$\begin{aligned}\mathrm{Compress}_d : \quad & \mathbb{Z}_q \longrightarrow \mathbb{Z}_{2^d} \\ & x \longrightarrow \lceil (2d/q) \cdot x \rfloor\end{aligned}$$

- **Feature:** Succinct, unambiguous, mathematical
- Uses mathematical integers, in principle unbounded
- Uses modular field arithmetic, with specific rounding functions
- ML-KEM also uses polynomials, vectors, matrices
- Other crypto standards use elliptic curves, finite fields, pairing-based curves, ...

Mathematical Algorithms

- Computes a math function
- Uses loops, variables
- Easy to implement
- Not so simple to understand

- Is this a “good” spec?
- Is it correct?
- **Desired Feature:**
  “We hold these specs to be self-evidently correct”

Algorithm 9 $\mathsf{NTT}^{-1}(\hat{f})$

Computes the polynomial $f \in R_q$ corresponding to the given NTT representation $\hat{f} \in T_q$.

Input: array $\hat{f} \in \mathbb{Z}_q^{256}$. ▷ the coefficients of input NTT representation

Output: array $f \in \mathbb{Z}_q^{256}$. ▷ the coefficients of the inverse-NTT of the input

1: $f \leftarrow \hat{f}$ ▷ will compute in-place on a copy of input array

2: $k \leftarrow 127$

3: **for** $(len \leftarrow 2;\ len \leq 128;\ len \leftarrow 2 \cdot len)$

4:     **for** $(start \leftarrow 0;\ start < 256;\ start \leftarrow start + 2 \cdot len)$

5:         $zeta \leftarrow \zeta^{\mathsf{BitRev}_7(k)} \bmod q$

6:         $k \leftarrow k - 1$

7:         **for** $(j \leftarrow start;\ j < start + len;\ j{+}{+})$

8:             $t \leftarrow f[j]$

9:             $f[j] \leftarrow t + f[j + len]$ ▷ steps 9-10 done modulo q

10:             $f[j + len] \leftarrow zeta \cdot (f[j + len] - t)$

11:         **end for**

12:     **end for**

13: **end for**

14: $f \leftarrow f \cdot 3303 \bmod q$ ▷ multiply every entry by $3303 \equiv 128^{-1} \bmod q$

15: **return** f

EasyCrypt Spec

$$\textsf{op as\_sint}(x : \textsf{Fq}) = \textbf{if } (q-1)\ /\ 2 < \textsf{asint } x \textbf{ then } \textsf{asint } x - q \textbf{ else } \textsf{asint } x.$$

$$\textsf{op compress}(d : \textsf{int}, x : \textsf{Fq}) : \textsf{int} = \textsf{round } (\textsf{asint } x * 2^d /_{\mathbb{R}}\ q)\ \%\ 2^d.$$

$$\textsf{op decompress}(d : \textsf{int}, x : \textsf{int}) : \textsf{Fq} = \textsf{inFq } (\textsf{round } (x * q /_{\mathbb{R}}\ 2^d)).$$

$$\begin{aligned}&\textsf{op invntt}(p : \textsf{poly}) = \textsf{Array256.init } (\textbf{fun } i \Rightarrow \ \textbf{let } ii = i\ /\ 2 \textbf{ in}\\ &\quad \textbf{if } i\ \%\ 2 = 0 \textbf{ then } \textstyle\sum_{j=0}^{127} \textsf{inv } (\textsf{inFq } 128) * p[2*j] * \textsf{zroot}^{-(2*\textsf{br}\ j+1)*ii}\\ &\qquad\qquad\qquad \textbf{else } \textstyle\sum_{j=0}^{127} \textsf{inv } (\textsf{inFq } 128) * p[2*j+1] * \textsf{zroot}^{-(2*\textsf{br}\ j+1)*ii}).\end{aligned}$$

- **Feature:** Machine Checked
- **Feature:** Basis for security proof for ML-KEM
- **Feature:** Basis for correctness proof for Jasmin implementation

- Close to the mathematical spec (easy to eyeball and to formally verify)
- Can this be in the NIST spec? Is it stable? Is it readable for programmers?

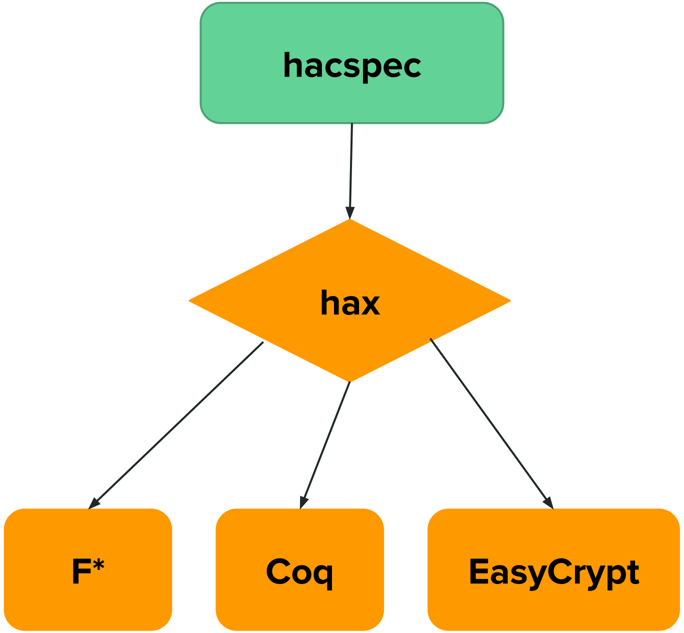
Python pseudocode in the IETF RFC

- Python, SAGE-friendly
- **Feature:** Executable
- **Feature:** Readable by programmers, written by cryptographers

- Is this a “good” spec?
- Is it correct?

```
Compress(x, d) = Round( (2^d / q) x ) umod 2^d

def InvNTT(self):
    cs = list(self.cs)
    layer = 2
    zi = n//2
    while layer < n:
        for offset in range(0, n-layer, 2*layer):
            zi -= 1
            z = pow(zeta, brv(zi), q)

            for j in range(offset, offset+layer):
                t = (cs[j+layer] - cs[j]) % q
                cs[j] = (inv2*(cs[j] + cs[j+layer])) % q
                cs[j+layer] = (inv2 * z * t) % q
        layer *= 2
    return Poly(cs)
```

An executable, translatable spec in hacspec




```
fn ntt_inverse(f_hat: KyberPolynomialRingElement) -> KyberPolynomialRingElement {
    let mut f = f_hat;
    let mut k: u8 = 127;
    // for (len <- 2; len <= 128; len <- 2*len)
    for len in NTT_LAYERS {
        // for (start <- 0; start < 256; start <- start + 2*len)
        for start in (0..(COEFFICIENTS_IN_RING_ELEMENT - len)).step_by(2 * len) {
            // zeta <- Zeta^(BitRev_7(k)) mod q
            let zeta = ZETA.pow(bit_rev_7(k));
            k -= 1;

            for j in start..start + len {
                let t = f[j];
                f[j] = t + f[j + len];
                f[j + len] = zeta * (f[j + len] - t);
            }
        }
    }
    // f <- f*3303 mod q
    for i in 0..f.coefficients().len() {
        f[i] = f[i] * INVERSE_OF_128;
    }
    f
}
```

Mathematical Precision vs. Implementation Guidance

- KyberSlash Attacks
- **Version 1:** timing attack due to division in Compress_1 applied to plaintext
- **Version 2:** timing attack due to division in Compress_12 applied to IND-CPA ciphertext
- Would having secrecy annotations in the spec have helped?

$$\begin{aligned} \mathrm{Compress}_d : \quad & \mathbb{Z}_q \longrightarrow \mathbb{Z}_{2^d} \\ & x \longrightarrow \lceil (2d/q) \cdot x \rfloor \end{aligned}$$

```
// t += ((int16_t)t >> 15) & KYBER_Q;
// t  = (((t << 1) + KYBER_Q/2)/KYBER_Q) & 1;
t <<= 1;
t += 1665;
t *= 80635;
t >>= 28;
t &= 1;
```

Specs for Constructions & Protocols

CryptoVerif (Signed DH, HPKE, WireGuard)

- Process calculus
- Defines protocol actions, cryptographic assumptions, security goals, as oracles,
- **Feature:** Machine-checked
- **Feature:** Close to pen-and-paper proofs written by cryptographers

- Should this be in the HPKE RFC?

```
let processA(hf:hashfunction, skA:skey) =
  OA1(hostX: host) :=
    a <-R Z;
    ga <- exp(g,a);
    return(A, hostX, ga);

  OA3(=A, =hostX, gb:G, s:signature) :=
    get keys(=hostX, pkX) in
    if verify(msg2(A, hostX, ga, gb), pkX, s) then
    gba <- exp(gb, a);
    kA <- hash(hf, gba);
    event endA(A, hostX, ga, gb);
    return(sign(msg3(A, hostX, ga, gb), skA));

  OAfin() :=
    if hostX = B then (
      keyA:key <- kA
    ) else
      return(kA).
```

ProVerif (TLS 1.3, Signal, ...)

- Process calculus
- Defines protocol actions, **symbolic** cryptographic assumptions, security goals, as concurrent processes
- **Feature:** Machine-checked
- **Feature:** Fully automatic, finds protocol flaws, MitM attacks

- Not a crypto proof (symbolic)
- Should this be in the TLS RFC?

```
(**********************************************************)
(*  TLS 1.3 0+1-RTT Processes: no client auth, uses psk (potentially NoPSK) *)
(**********************************************************)


let Client13() =
    (get preSharedKeys(a,b,psk) in
     in (io,ioffer:params);
     let nego(=TLS13,DHE_13(g,eee),hhh,aaa,pt) = ioffer in
     new cr:random;
     let (x:bitstring,gx:element) = dh_keygen(g) in
     let (early_secret:bitstring,kb:mac_key) = kdf_es(psk) in
     let zoffer = nego(TLS13,DHE_13(g,gx),hhh,aaa,Binder(zero)) in
     let pt = Binder(hmac(StrongHash,kb,msg2bytes(CH(cr,zoffer)))) in
     let offer = nego(TLS13,DHE_13(g,gx),hhh,aaa,pt) in
     let ch = CH(cr,offer) in
     event ClientOffersVersion(cr,TLS13);
     event ClientOffersKEX(cr,DHE_13(g,gx));
     event ClientOffersAE(cr,aaa);
     event ClientOffersHash(cr,hhh);
     out(io,ch);
     let (kc0:ae_key,ems0:bitstring) = kdf_k0(early_secret,msg2bytes(ch)) in
     insert clientSession0(cr,psk,offer,kc0,ems0);

     in(io,SH(sr,mode));
     let nego(=TLS13,DHE_13(=g,gy),h,a,spt) = mode in
     let log = (ch,SH(sr,mode)) in

     let gxy = e2b(dh_exp(g,gy,x)) in
     let handshake_secret = kdf_hs(early_secret,gxy) in
     let (master_secret:bitstring,chk:ae_key,shk:ae_key,cfin:mac_key,sfin:mac_key) =
```

Questions: what makes a good spec?

Questions for discussion

- Should we embed formal specifications within NIST and IETF crypto standards?
- If not, would it be possible to link the pseudocode used in these standards with formal specifications?
- Is it more valuable to have an executable specification for testing or a formal spec for verification?
- Are specifications written in languages like Python and Rust more accessible, readable, usable than specifications written in formal languages like F* or EasyCrypt?
- Should formal specifications describe high-level mathematical concepts like polynomial multiplication or should they detail low-level algorithms like NTT multiplication?
- Should specifications in standards be targeted towards security proofs or implementation correctness, and can they do both?
- Should standards and their formal specifications include indications for secure implementations, such as algorithms that may be at risk of side-channel attacks?

hacspec

hacspec: a tool-independent spec language

Design Goals

- **Easy to use** for crypto developers
- **Familiar** language and tools
- **Succinct** specs, like pseudocode
- **Strongly typed** to avoid spec errors
- **Executable** for spec debugging
- **Testable** against RFC test vectors
- **Translations** to formal languages like

  F*, Coq, EasyCrypt, ...

hacspec: a tool-independent spec language

Design Goals

- **Easy to use** for crypto developers
- **Familiar** language and tools
- **Succinct** specs, like pseudocode
- **Strongly typed** to avoid spec errors
- **Executable** for spec debugging
- **Testable** against RFC test vectors
- **Translations** to formal languages like
  F*, Coq, EasyCrypt, ...

A purely functional subset of Rust

- Safe Rust without external side-effects
- No mutable borrows
- All values are copyable
- Rust tools & development environment
- A library of common abstractions
  - Arbitrary-precision Integers
  - Secret-independent Machine Ints
  - Vectors, Matrices, Polynomials,...

Language and Toolchain Details: hacspec.org

hacspec: purely functional crypto code in Rust

```
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 RFC

Call-by-value

```rust
fn inner_block(st: State) -> State {
   let mut state = st;
   state = chacha20_quarter_round(0, 4, 8, 12, state);
   state = chacha20_quarter_round(1, 5, 9, 13, state);
   state = chacha20_quarter_round(2, 6, 10, 14, state);
   state = chacha20_quarter_round(3, 7, 11, 15, state);
   state = chacha20_quarter_round(0, 5, 10, 15, state);
   state = chacha20_quarter_round(1, 6, 11, 12, state);
   state = chacha20_quarter_round(2, 7, 8, 13, state);
   chacha20_quarter_round(3, 4, 9, 14, state)
}
```

State-passing style

ChaCha20 in hacspec

hacspec: abstract integers for field arithmetic

```
n = le_bytes_to_num(msg[((i-1)*16)..(i*16)] | [0x01])
a += n
a = (r * a) % p
```

Poly1305 RFC (update_block)

Modular 130-bit Prime Field Arithmetic

```
pub fn poly1305_encode_block(b: &PolyBlock) -> FieldElement {
    let n = U128_from_le_bytes(U128Word::from_seq(b));
    let f = FieldElement::from_secret_literal(n);
    f + FieldElement::pow2(128)
}

pub fn poly1305_update_block(b: &PolyBlock, (acc,r,s): PolyState) -> PolyState {
    ((poly1305_encode_block(b) + acc) * r, r, s)
}
```

Poly1305 in hacspec

Modular Arithmetic over User-Defined Field

hacspec: secret integers for “constant-time” code

Separate Secret and Public Values

- New types: U8, U32, U64, U128
- Can do arithmetic: +, *, -
- Can do bitwise ops: ^, |, &
- Cannot do division: /, %
- Cannot do comparison: ==, !=, <, ...
- Cannot use as array indexes: x[u]

Enforces secret independence

- A “constant-time” discipline
- Important for some crypto specs

```
fn chacha20_line(a: StateIdx, b: StateIdx, d: StateIdx,
                 s: usize, mut state: State) -> State {
    state[a] = state[a] + state[b];
    state[d] = state[d] ^ state[a];
    state[d] = state[d].rotate_left(s);
    state
}
```

ChaCha20 in hacspec

```
fn sub_bytes(state: Block) -> Block {
    let mut st = state;
    for i in 0..BLOCKSIZE {
        st[i] = SBOX[U8::declassify(state[i])];
    }
    st
}
```

AES in hacspec

hacspec: translation to formal languages

ChaCha20 in hacspec

```
pub fn chacha20_quarter_round(
    a: StateIdx,
    b: StateIdx,
    c: StateIdx,
    d: StateIdx,
    mut state: State,
) -> State {
    state = chacha20_line(a, b, d, 16, state);
    state = chacha20_line(c, d, b, 12, state);
    state = chacha20_line(a, b, d, 8, state);
    chacha20_line(c, d, b, 7, state)
}
```

F* Spec

```
let chacha20_quarter_round (a b c d: state_idx_t) (state: state_t) : state_t =
  let state:state_t = chacha20_line a b d 16 state in
  let state:state_t = chacha20_line c d b 12 state in
  let state:state_t = chacha20_line a b d 8 state in
  chacha20_line c d b 7 state
```

Coq Spec

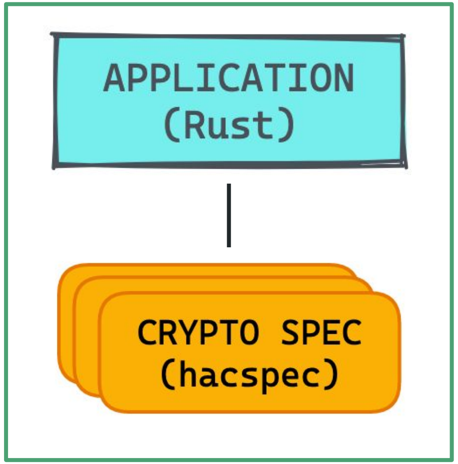
```
Definition chacha20_quarter_round (a : int32) (b : int32) (c : int32)
                                  (d : int32) (state : State) : State :=
  let state := chacha20_line a b d 16 state : State in
  let state := chacha20_line c d b 12 state : State in
  let state := chacha20_line a b d 8 state : State in
  chacha20_line c d b 7 state.
```

EasyCrypt Spec

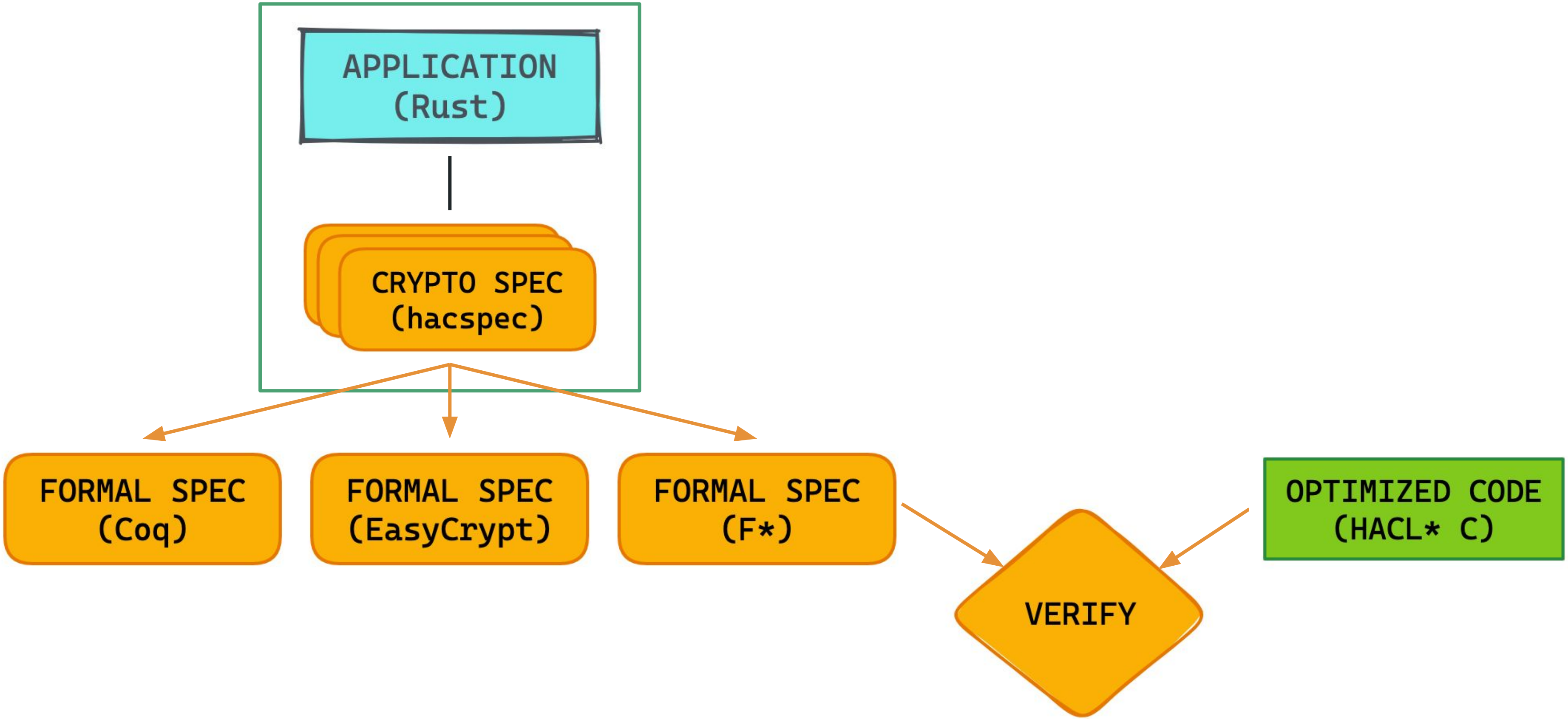
```
proc chacha20_quarter_round(a : int, b : int, c : int, d : int,
                            state : State) = {
  var _res;
  state <@ chacha20_line (a, b, d, 16, state);
  state <@ chacha20_line (c, d, b, 12, state);
  state <@ chacha20_line (a, b, d, 8, state);
  _res <@ chacha20_line (c, d, b, 7, state);
  return _res;
}
```

Active development: github.com/hacspec

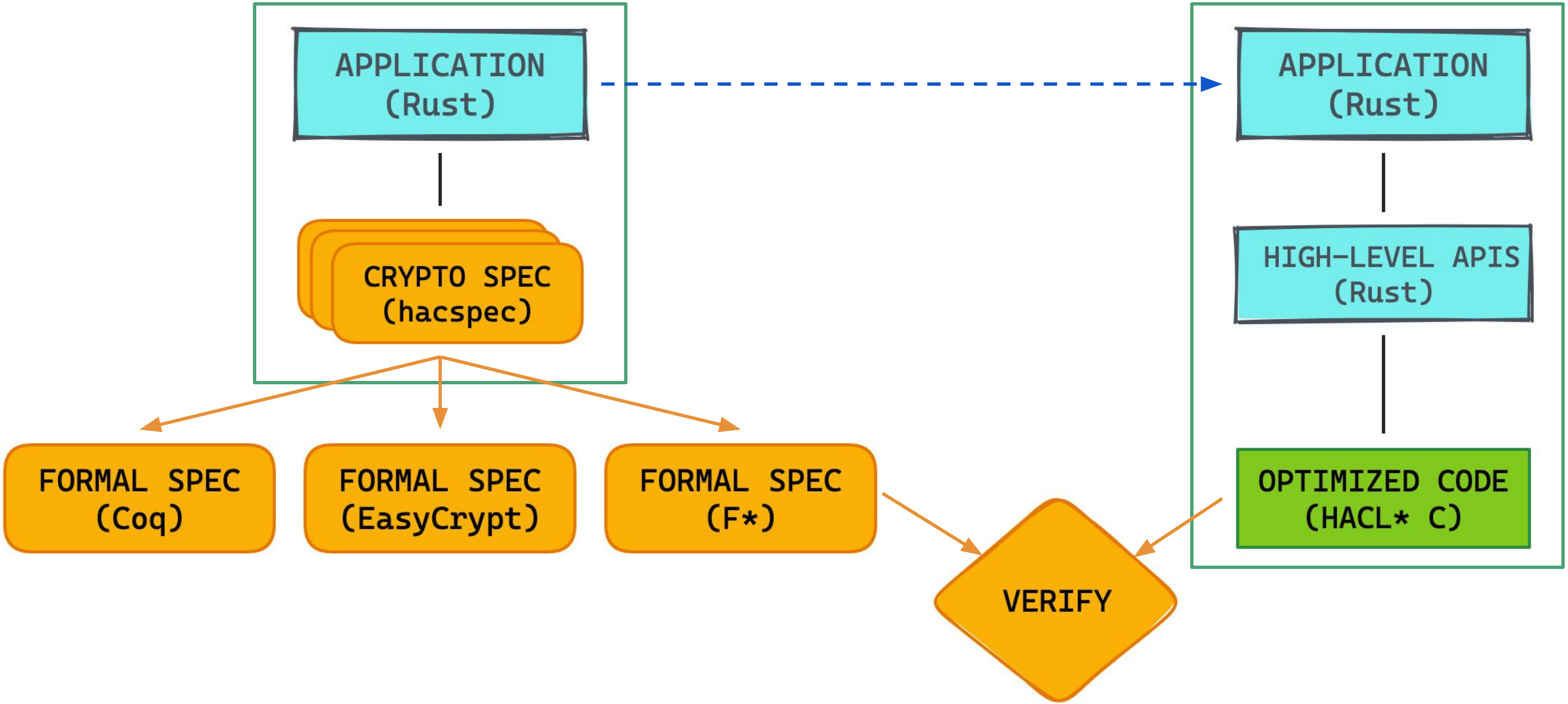
hacspec: towards high-assurance crypto software



hacspec: towards high-assurance crypto software

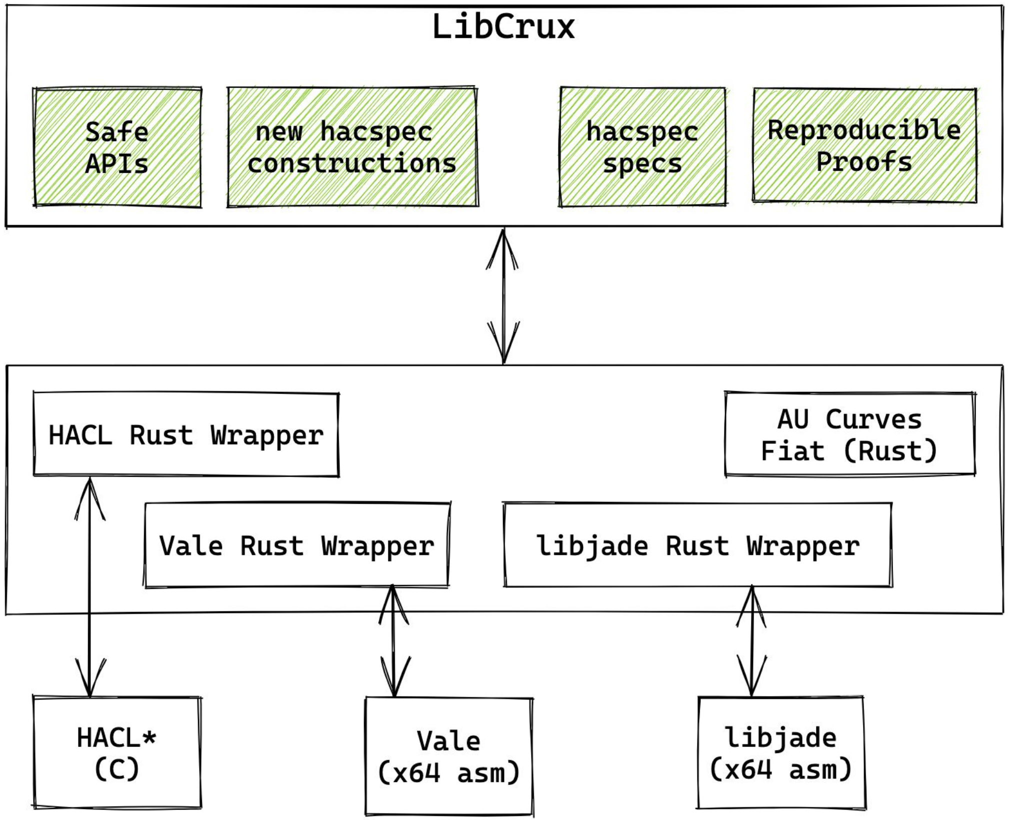


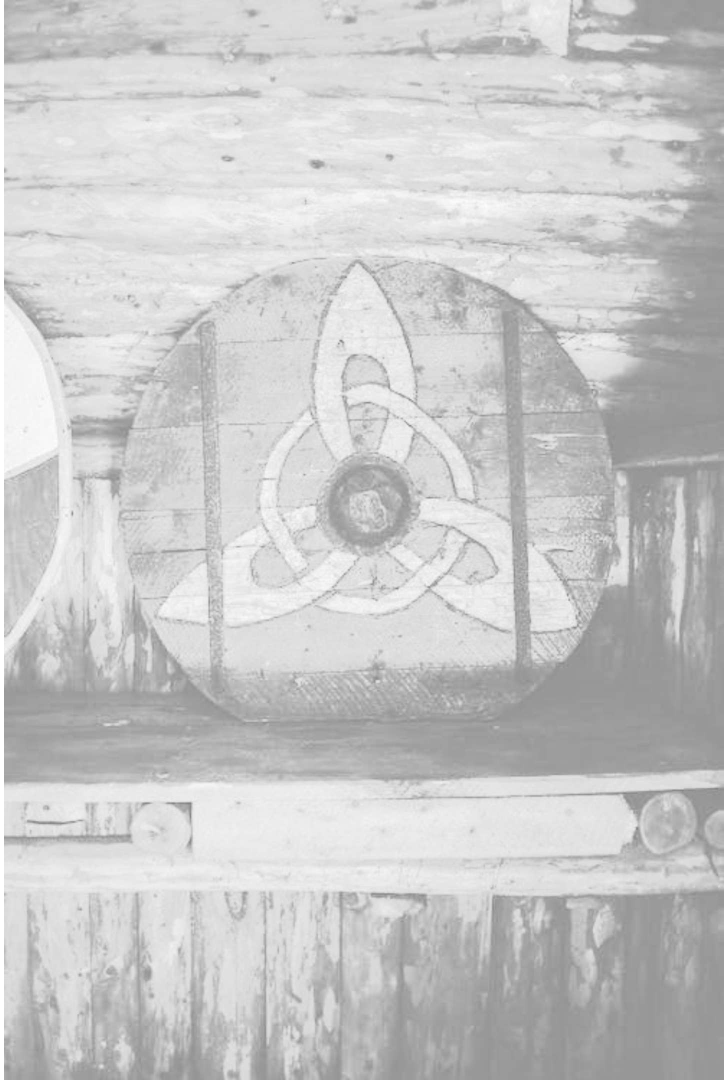
hacspec: towards high-assurance crypto software



libcrux: a library of verified cryptography

libcrux: architecture





Unsafe APIs: Array Constraints

```
void
Hacl_Chacha20Poly1305_32_aead_encrypt(
  uint8_t *k,
  uint8_t *n,
  uint32_t aadlen,
  uint8_t *aad,
  uint32_t mlen,
  uint8_t *m,
  uint8_t *cipher,
  uint8_t *mac
);
```

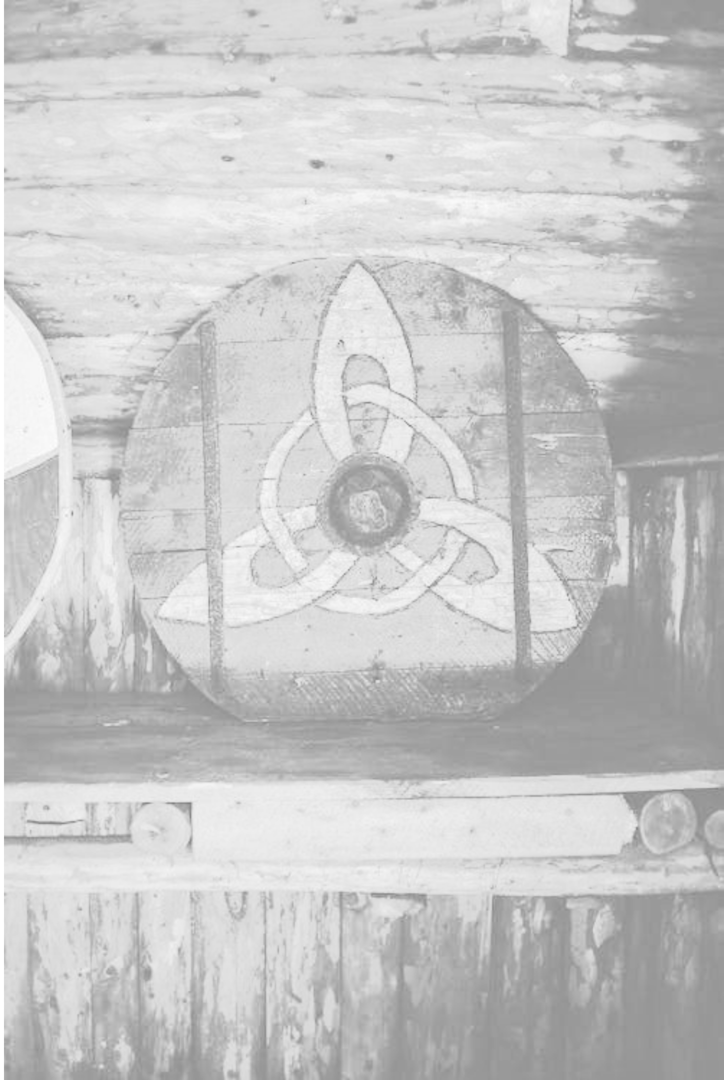
Fixed Length

Disjoint

Verified F* API: Preconditions

```
let aead_encrypt_st (w:field_spec) =
    key:lbuffer uint8 32ul
  -> nonce:lbuffer uint8 12ul
  -> alen:size_t
  -> aad:lbuffer uint8 alen
  -> len:size_t
  -> input:lbuffer uint8 len
  -> output:lbuffer uint8 len
  -> tag:lbuffer uint8 16ul ->
  Stack unit
  (requires fun h ->
    live h key /\ live h nonce /\ live h aad /\
    live h input /\ live h output /\ live h tag /\
    disjoint key output /\ disjoint nonce output /\
    disjoint key tag /\ disjoint nonce tag /\
    disjoint output tag /\ eq_or_disjoint input output /\
    disjoint aad output)
```

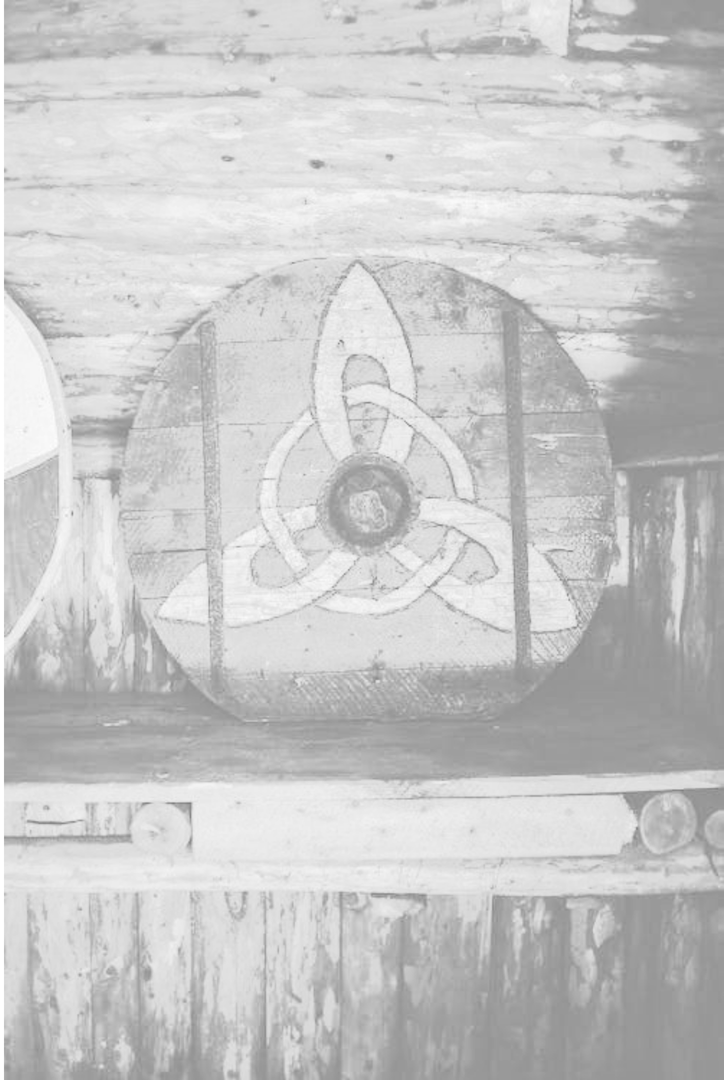
Length Constraints



Verified F* API: Preconditions

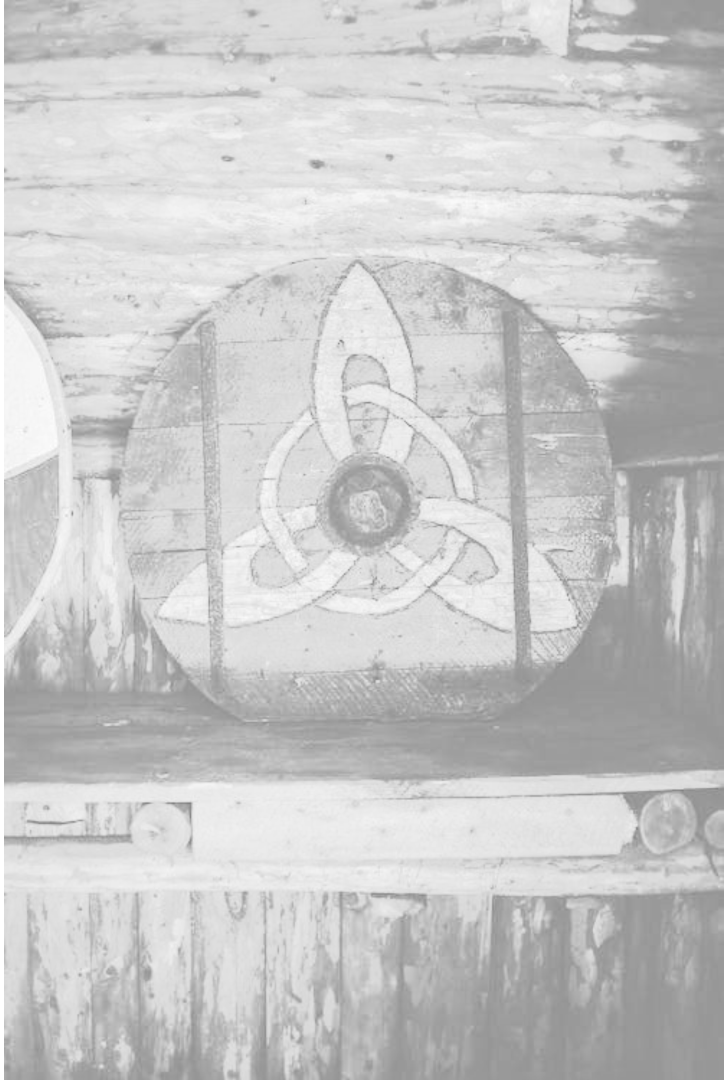
```
let aead_encrypt_st (w:field_spec) =
    key:lbuffer uint8 32ul
  -> nonce:lbuffer uint8 12ul
  -> alen:size_t
  -> aad:lbuffer uint8 alen
  -> len:size_t
  -> input:lbuffer uint8 len
  -> output:lbuffer uint8 len
  -> tag:lbuffer uint8 16ul ->
  Stack unit
  (requires fun h ->
    live h key /\ live h nonce /\ live h aad /\
    live h input /\ live h output /\ live h tag /\
    disjoint key output /\ disjoint nonce output /\
    disjoint key tag /\ disjoint nonce tag /\
    disjoint output tag /\ eq_or_disjoint input output /\
    disjoint aad output)
```

Disjointness Constraints



libcrux: Typed Rust APIs

```rust
type Chacha20Key = [u8; 32];
type Nonce = [u8; 12];
type Tag = [u8; 16];

fn encrypt(
    key: &Chacha20Key,
    msg_ctxt: &mut [u8],
    nonce: Nonce,
    aad: &[u8]
) -> Tag
```



libcrux: supported algorithms & perf

Crypto Standard	Platforms	Specs	Implementations
ECDH • x25519 • P256	 Portable + Intel ADX Portable	 hacspec, F* hacspec, F*	 HACL*, Vale HACL*
AEAD • Chacha20Poly1305 • AES-GCM	 Portable + Intel/ARM SIMD Intel AES-NI	 hacspec, F*, EasyCrypt hacspec, F*	 HACL*, libjade Vale
Signature • Ed25519 • ECDSA P256 • BLS12-381	 Portable Portable Portable	 hacspec, F* hacspec, F* hacspec, Coq	 HACL* HACL* AUCurves
Hash • Blake2 • SHA2 • SHA3	 Portable + Intel/ARM SIMD Portable Portable + Intel SIMD	 hacspec, F* hacspec, F* hacspec, F*, EasyCrypt	 HACL* HACL* HACL*, libjade
HKDF, HMAC	Portable	hacspec, F*	HACL*
HPKE	Portable	hacspec	hacspec

libcrux: performance

	libcrux	Rust Crypto	Ring	OpenSSL
Sha3 256	574.39 MiB/s	573.89 MiB/s	unsupported	625.37 MiB/s
x25519	30.320 μs	35.465 μs	30.363 μs	32.272 μs

libjade

HACL* + Vale

Intel Kaby Lake (ADX, AVX2)

	libcrux	Rust Crypto	Ring	OpenSSL
Sha3 256	337.67 MiB/s	275.05 MiB/s	unsupported	322.21 MiB/s
x25519	37.640 μs	67.660 μs	71.236 μs	48.620 μs

HACL*

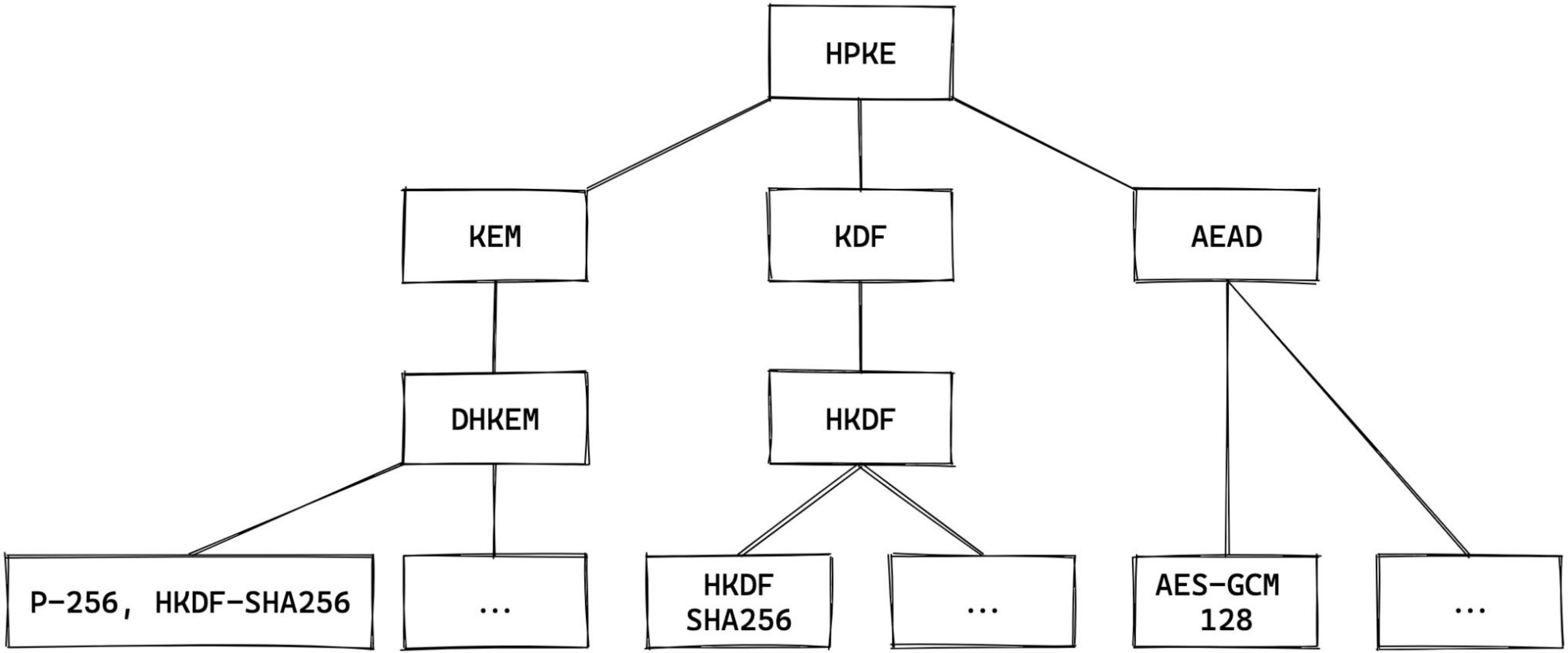
Apple Arm M1 Pro (Neon)

Stream:	Internet Research Task Force (IRTF)
RFC:	9180
Category:	Informational
Published:	February 2022
ISSN:	2070-1721
Authors:	R. Barnes *Cisco* · K. Bhargavan *Inria* · B. Lipp *Inria* · C. Wood *Cloudflare*

RFC 9180
Hybrid Public Key Encryption

https://tech.cryspen.com/hpke-spec/hpke/index.html

HPKE: Construction



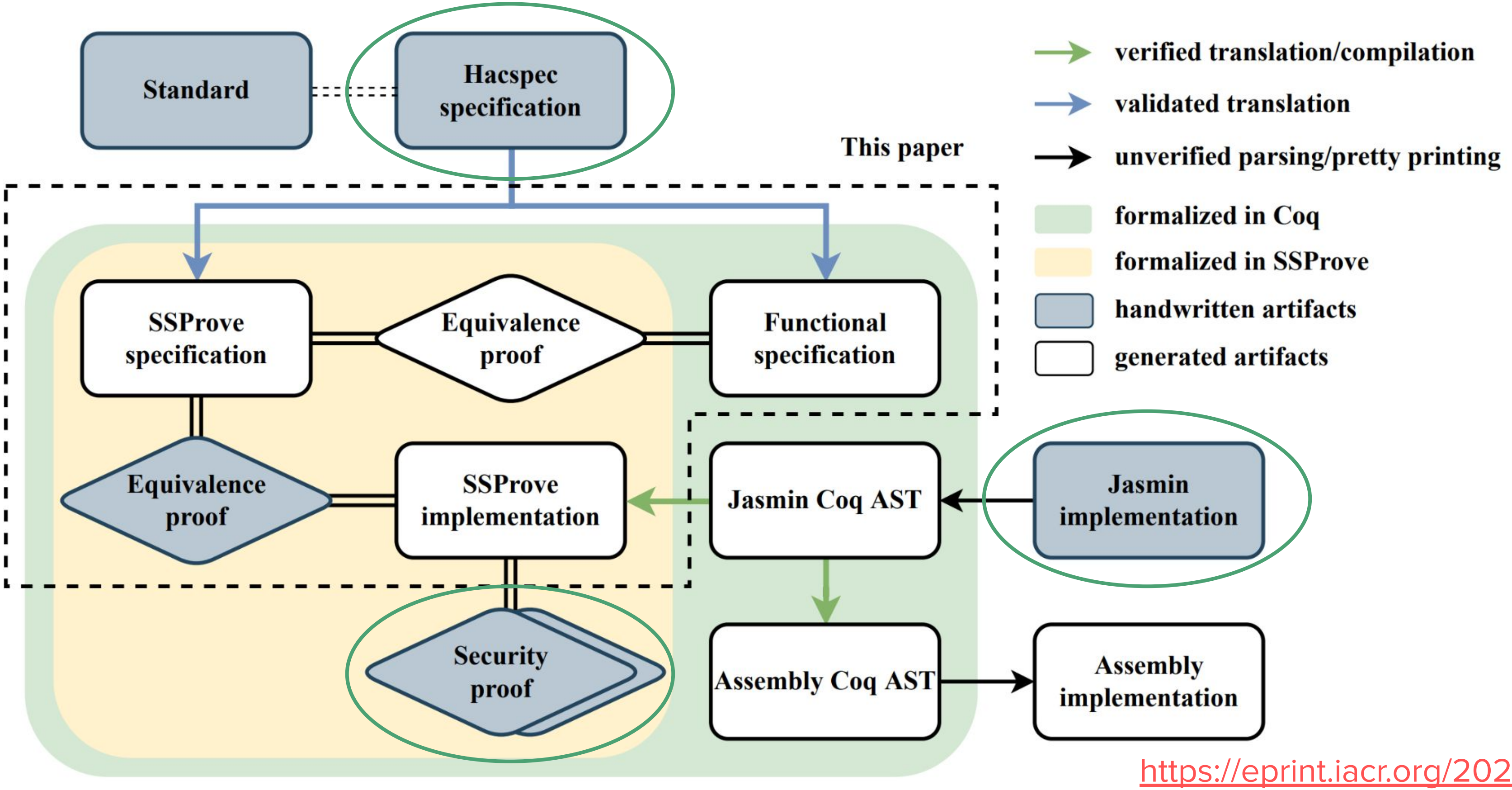
HPKE code performance: hacspec vs. stateful Rust

	hacspec HPKE	Rust HPKE
Setup Sender	79.9 µs	68 µs
Setup Receiver	76 µs	54.4 µs

	libcrux	RustCrypto
Sha2 256	311.76 MiB/s	319.10 MiB/s
x25519	30.320 µs	35.465 µs
x25519 base	30.218 µs	11.812 µs
ChaCha20Poly1305	758.89 MiB/s	249.33 MiB/s

Ongoing and Future Work

The Last Yard: linking hacspec to security proofs



https://eprint.iacr.org/2023/185

Verification Tools: more proof backends for hacspec

Security Analysis Tools

- SSProve: modular crypto proofs
- EasyCrypt: verified constructions

- ProVerif: symbolic protocol proofs
- CryptoVerif: verified protocols
- Squirrel: protocol verifier

Program Verification Tools

- QuickCheck: logical spec testing
- Creusot: verifying spec contracts
- Aeneas: verifying Rust code

- LEAN: verification framework
- <Your favourite prover here>

Conclusions

- **Fast verified code** is available today for most modern crypto algorithms
  - + some post-quantum crypto; Future: verified code for ZKP, FHE, MPC, …
  - Most code in C or Intel assembly; Ongoing: Rust, ARM assembly, …
- **hacspec** can be used as a common spec language for multiple tools/libraries
  - Ongoing: adding new Rust features, new proof backends, linking with Rust verifiers, …
  - **Try it yourself:** hacspec.org
- **libcrux** provides safe Rust APIs to multiple verified crypto libraries
  - Ongoing: recipes for integrating new verified crypto from various research projects
  - **Try it yourself:** libcrux.org

Thanks!

- HACL*: https://github.com/hacl-star/hacl-star
- Vale: https://github.com/ValeLang/Vale
- libjade: https://github.com/formosa-crypto/libjade
- AUCurves: https://github.com/AU-COBRA/AUCurves

- hacspec: https://github.com/hacspec/hacspec
- libcrux: https://github.com/cryspen/libcrux