towards high assurance cryptographic software

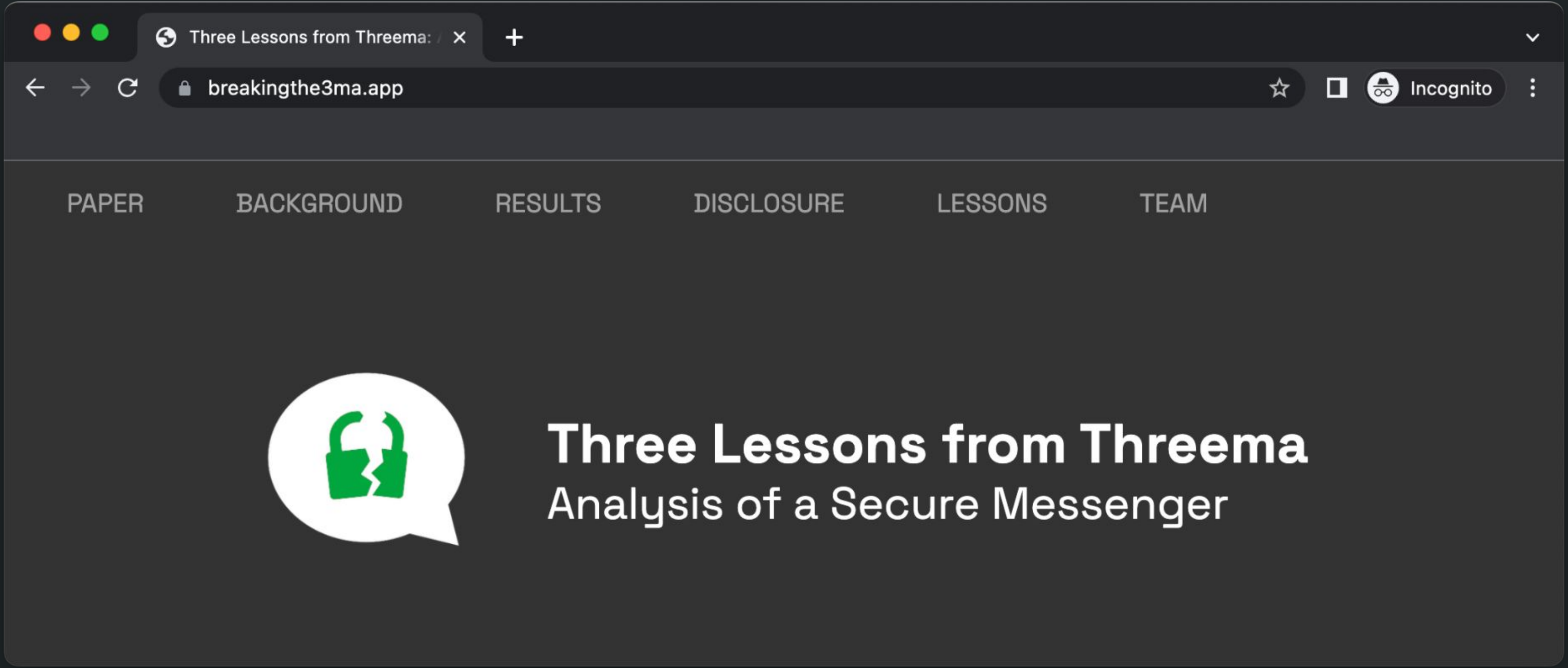
Karthikeyan Bhargavan

Joint work with many others at Inria, Cryspen, MSR, U. Porto, U. Aarhus, and elsewhere

The Golden Age of Crypto?

New Constructions	Post-Quantum Crypto, Homomorphic Encryption, …
New Protocols	Group Messaging, IoT Software Updates, …
New Applications	Blockchains, Privacy-Preserving Machine Learning, …

Designing Secure Protocols Is Still Hard



https://breakingthe3ma.app

Three Lessons From Threema: Analysis of a Secure Messenger

Kenneth G. Paterson
Applied Cryptography Group, ETH Zurich

Matteo Scarlata
Applied Cryptography Group, ETH Zurich

Kien Tuong Truong
Applied Cryptography Group, ETH Zurich

Usenix 2023

- Don’t roll your own crypto protocol
- Beware of cross-protocol interactions
- Formally analyze your protocol design using a (semi-automated) verification tool

Formal Analysis during Protocol Design

Crypto Constructions	HPKE, …
Crypto Protocols	TLS 1.3, MLS, …

The TLS 1.3 experiment [2014-2018]

Multi-year effort to redesign IETF Transport Layer Security

- 4 years, 28 drafts, 12 IETF meetings

Major contributions from academic security researchers

- **Cryptographic analyses and proofs (of drafts 5,9,10)**
  [Dowling et al. CCS'15-J.Crypt 2021, Jager et al. CCS'15, Krawczyk et al. Euro S&P'16, ...]
- **Mechanized cryptographic proofs (of draft 18) with CryptoVerif**
  [Bhargavan et al. S&P'17]
- **Automated symbolic protocol analysis with Tamarin and ProVerif**
  [Cremers et al. Oakland'16 + CCS'17, Bhargavan et al. S&P'17 + **CCS'22**]
- **Verified implementation code in F***
  [Bhargavan et al. S&P'17 and S&P'17]

The TLS 1.3 experiment [2014-2018]

Multi-year effort to redesign IETF Transport Layer Security

- 4 years, 28 drafts, 12 IETF meetings

Major contributions from academic security researchers

ACM CCS 2023

A Symbolic Analysis of Privacy for TLS 1.3 with Encrypted Client Hello

Karthikeyan Bhargavan
Inria Paris
Paris, France
karthikeyan.bhargavan@inria.fr

Vincent Cheval
Inria Paris
Paris, France
vincent.cheval@inria.fr

Christopher Wood
Cloudflare
San Francisco, United States
chriswood@cloudflare.com

How to encrypt a message?

Encrypt a symmetric key using a peer's public key

- Standard PKE/KEM. Just use RSA, ECIES, PQ-KEM?

Long messages? A stream of messages?

Sender authentication? PKE with Associated Data?

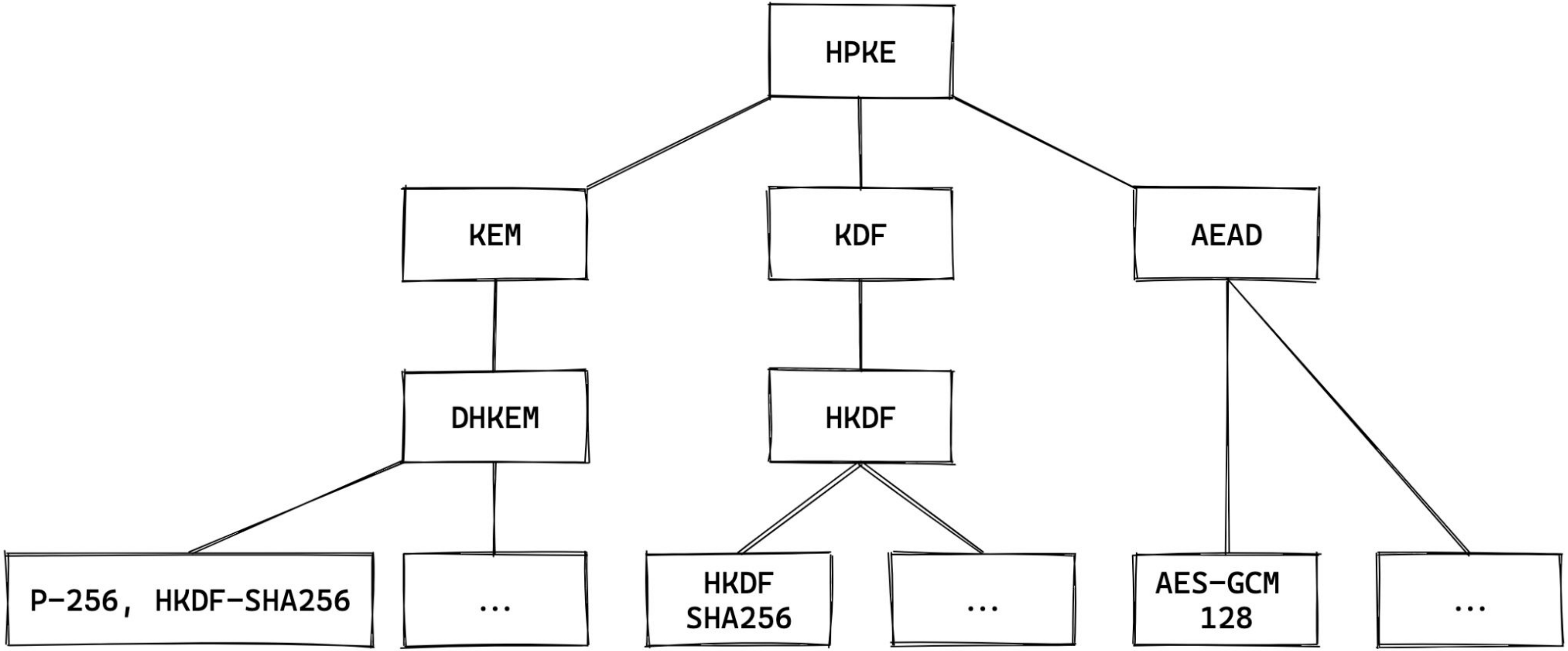
Stream:	Internet Research Task Force (IRTF)			
RFC:	9180			
Category:	Informational			
Published:	February 2022			
ISSN:	2070-1721			

Authors:	R. Barnes	K. Bhargavan	B. Lipp	C. Wood
	Cisco	*Inria*	*Inria*	*Cloudflare*

RFC 9180
Hybrid Public Key Encryption

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

HPKE: Agile, Modular Construction



HPKE Proof using CryptoVerif

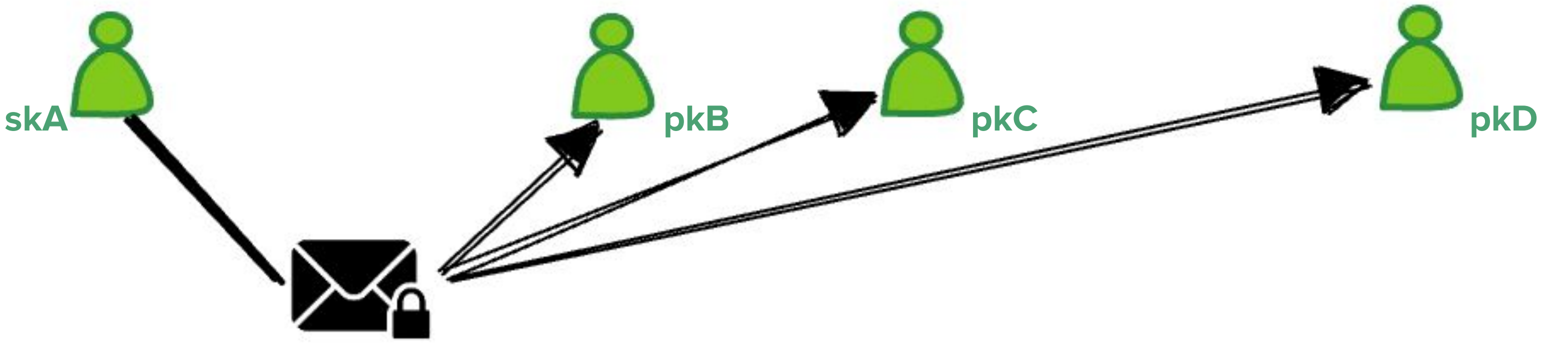
EuroCrypt 2022

Analysing the HPKE Standard *

Joël Alwen[1], Bruno Blanchet[2], Eduard Hauck[3], Eike Kiltz[3], Benjamin Lipp[2], and Doreen Riepel[3]

[1] Wickr
jalwen@wickr.com
[2] Inria Paris
{bruno.blanchet,benjamin.lipp}@inria.fr
[3] Ruhr-Universität Bochum
{eduard.hauck,eike.kiltz,doreen.riepel}@rub.de

How to encrypt group messages?

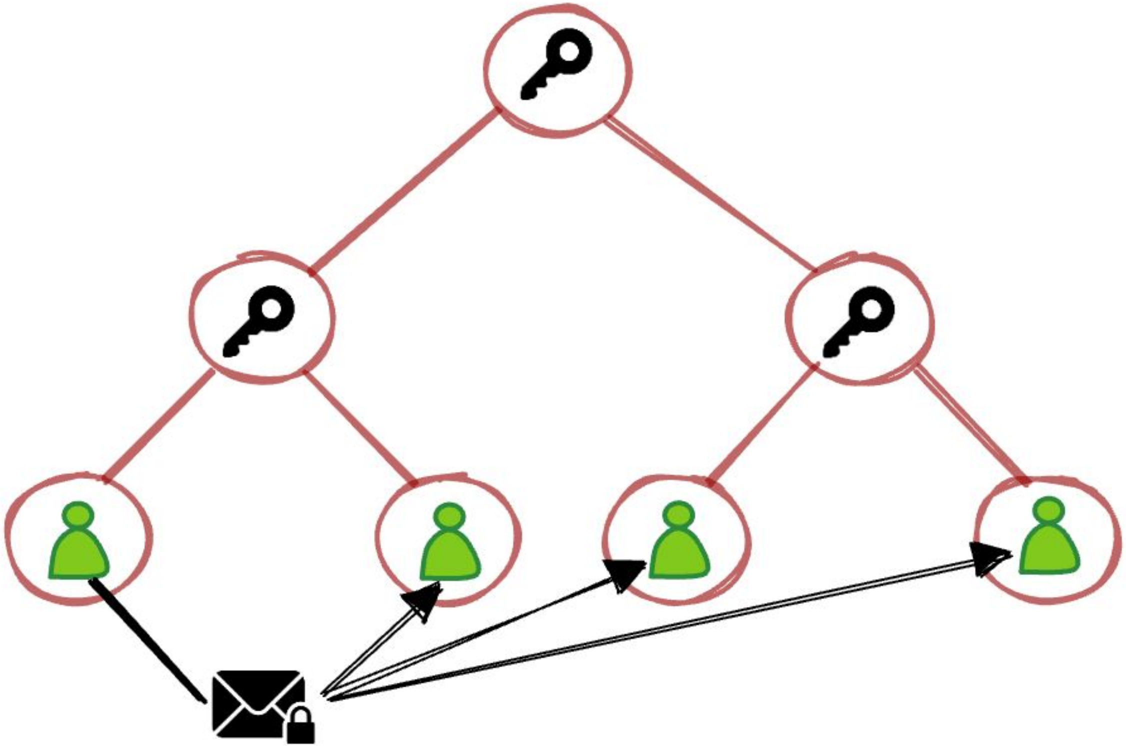


Use HPKE to encrypt message N times?

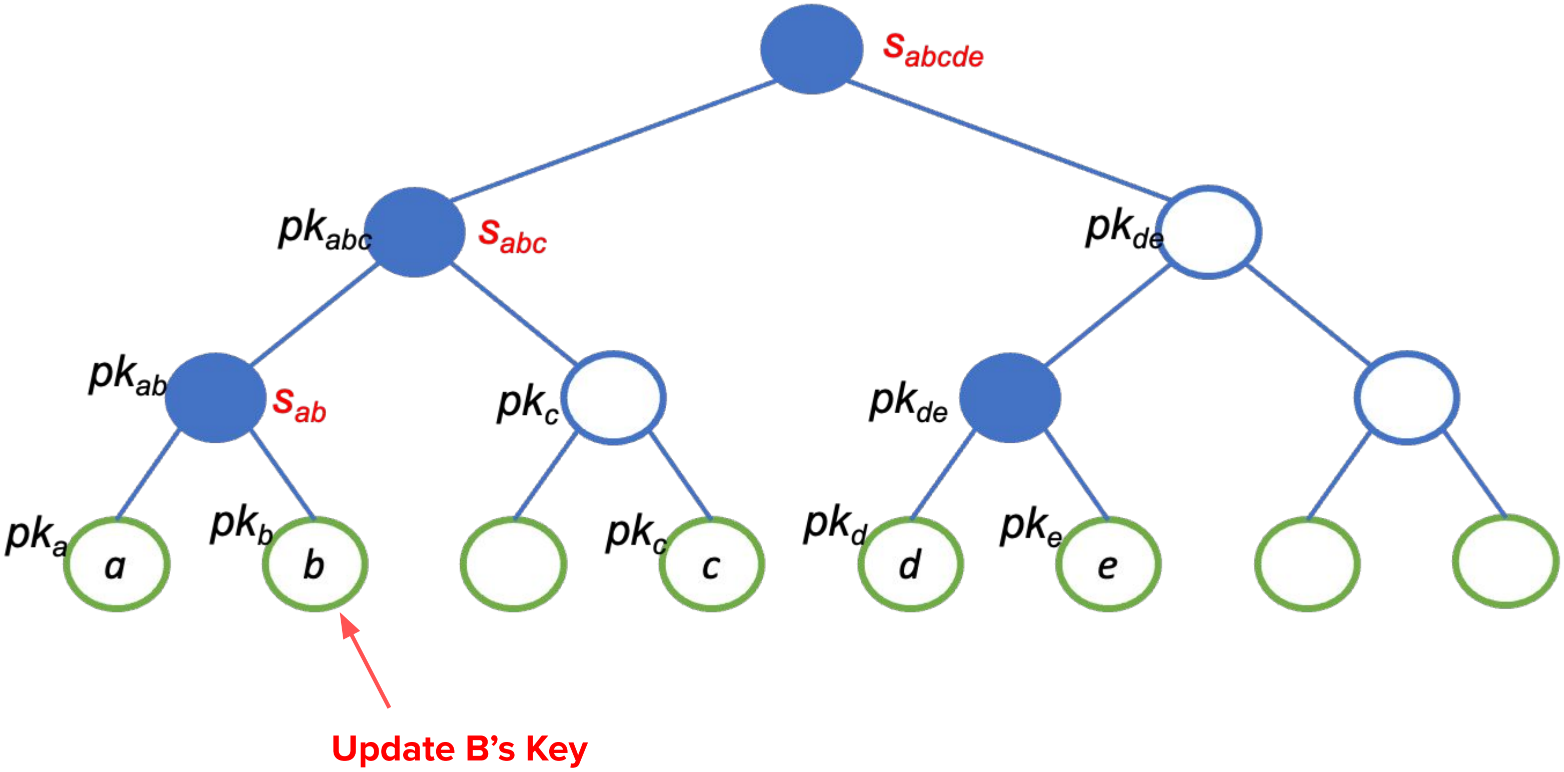
- **Sender Keys:** KEM key to N recipients, Sign every message
- O(N) computation for key changes

TreeKEM: keys for a tree of subgroups

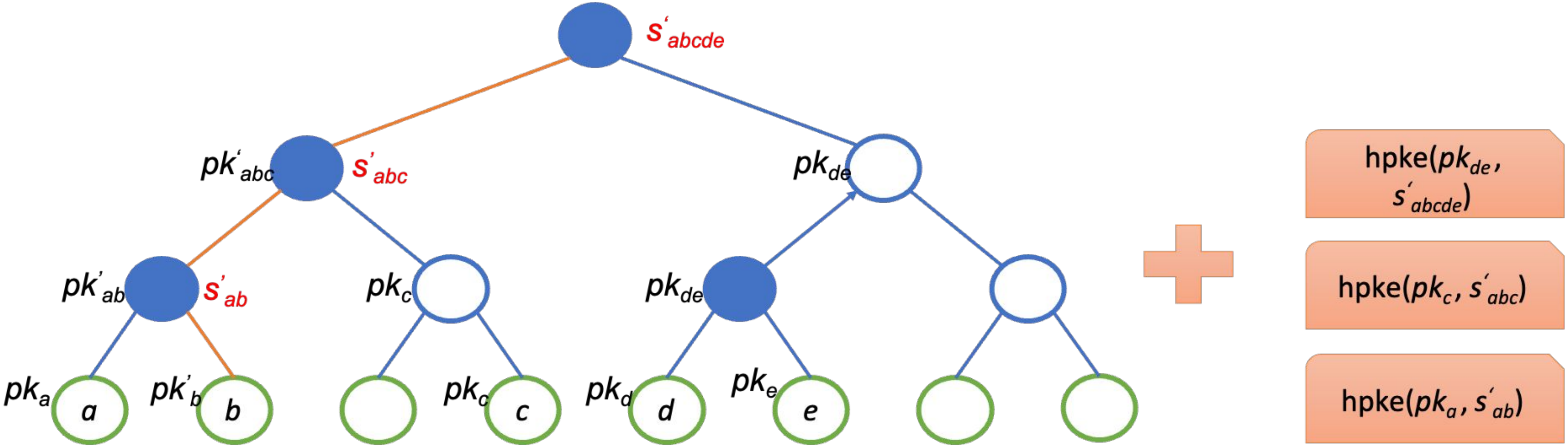
[Bhargavan, Barnes, Rescorla, 2018]



TreeKEM: keys for a tree of subgroups

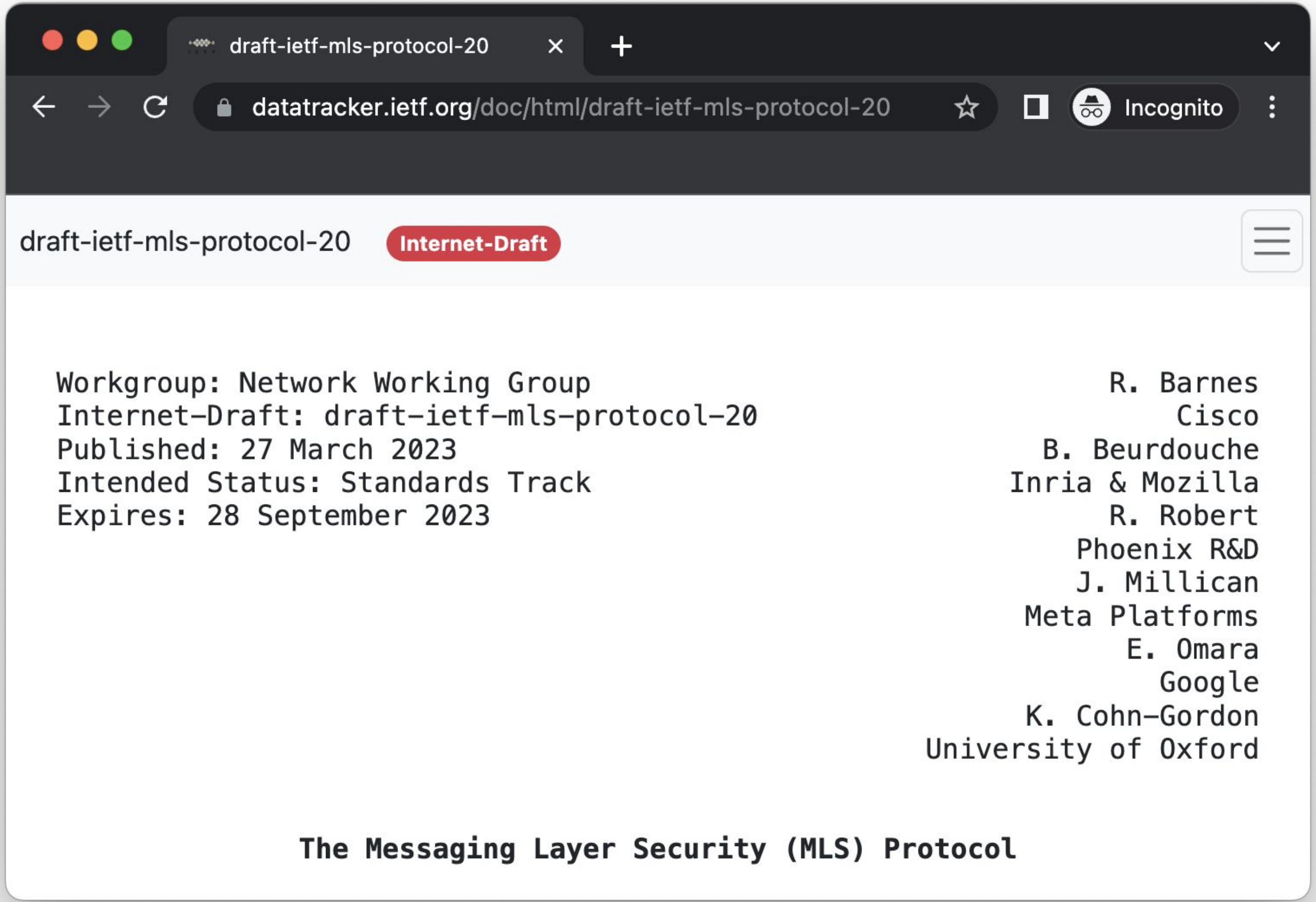


TreeKEM: keys for a tree of subgroups



Maintain a tree of subgroup keys with efficient updates

- Add is O(1), Remove and Update are O(log n)



Workgroup: Network Working Group
Internet-Draft: draft-ietf-mls-protocol-20
Published: 27 March 2023
Intended Status: Standards Track
Expires: 28 September 2023

R. Barnes
Cisco
B. Beurdouche
Inria & Mozilla
R. Robert
Phoenix R&D
J. Millican
Meta Platforms
E. Omara
Google
K. Cohn-Gordon
University of Oxford

The Messaging Layer Security (MLS) Protocol

Decomposing Messaging Layer Security

TreeSync	**synchronize membership and tree**
+	
TreeKEM	**derive, encapsulate subgroup keys**
+	
TreeDEM	**encrypt application messages**

Decomposing Messaging Layer Security

TreeSync	**synchronize membership and tree** (authentication, integrity)	TREE HASH + SIG
+		
TreeKEM	**derive, encapsulate subgroup keys** (forward secrecy, post-compromise security)	HPKE + KDF
+		
TreeDEM	**encrypt application messages** (forward secrecy, sender auth)	KDF + AEAD + SIG

Decomposing Messaging Layer Security

TreeSync | synchronize membership and tree (authentication) | TREE HASH + SIG

TreeSync: Authenticated Group Management for Messaging Layer Security

Théophile Wallez
Inria Paris

Jonathan Protzenko
Microsoft Research

Benjamin Beurdouche
Mozilla

Karthikeyan Bhargavan
Inria Paris

Usenix 2023

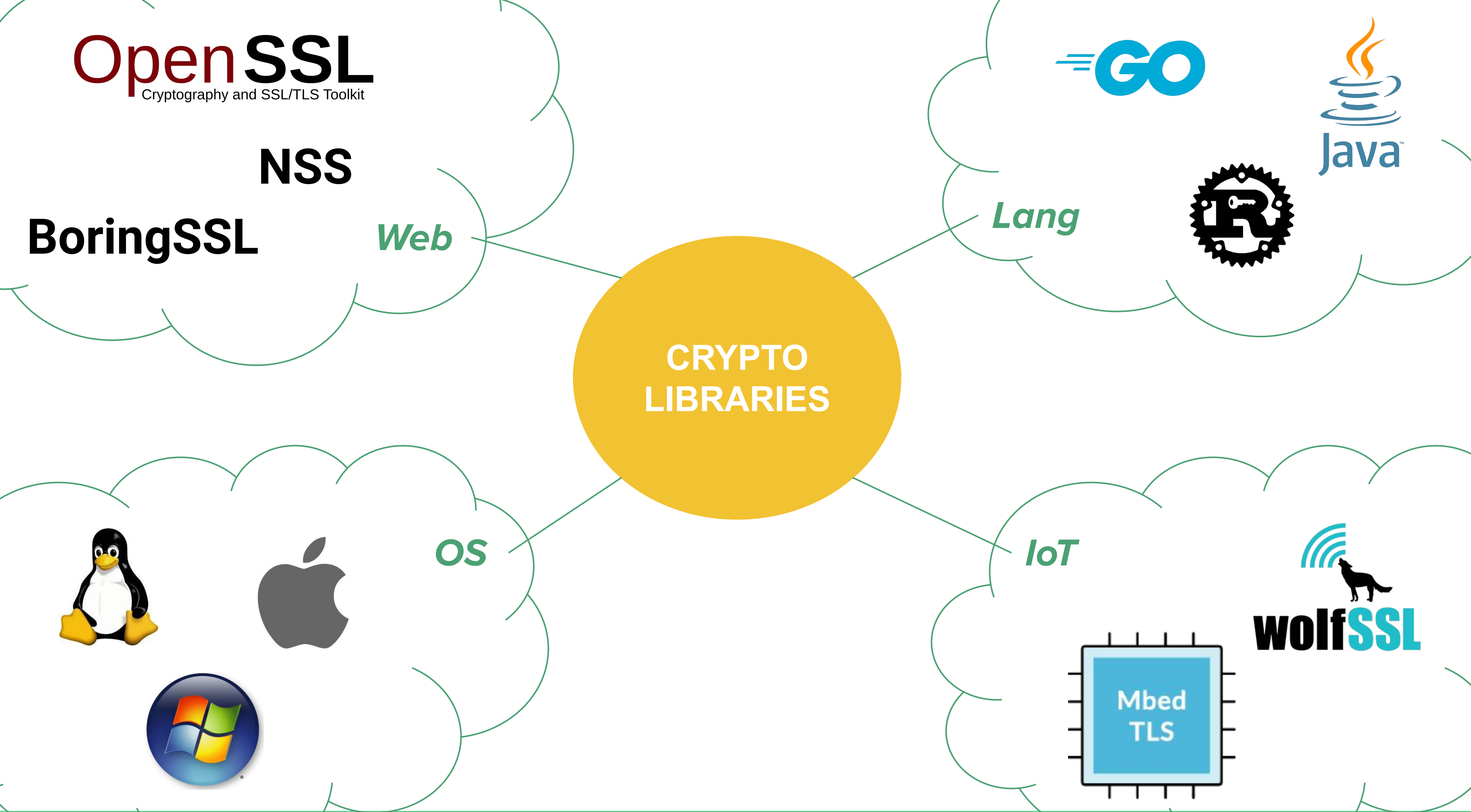
Many Ongoing Analyses of IETF MLS

	MLS Version	Part Analyzed	Adversarial Model	Considers Group Splits	Framework
[25]	Draft 1 (ART)	CGKA in static groups	active	yes	part game-based, part symbolic
[6]	Draft 6	CGKA	passive	no	game-based
[18]	Draft 7	Messaging	insider	yes	symbolic
[7]	Draft 11	Messaging	semi-active	yes	game-based
[20]	Draft 11	Key derivation	insider	n/a	game-based
[26]	Draft 11	Multi-group messaging	n/a	n/a	n/a
this work	Draft 12	CGKA	insider	yes	UC

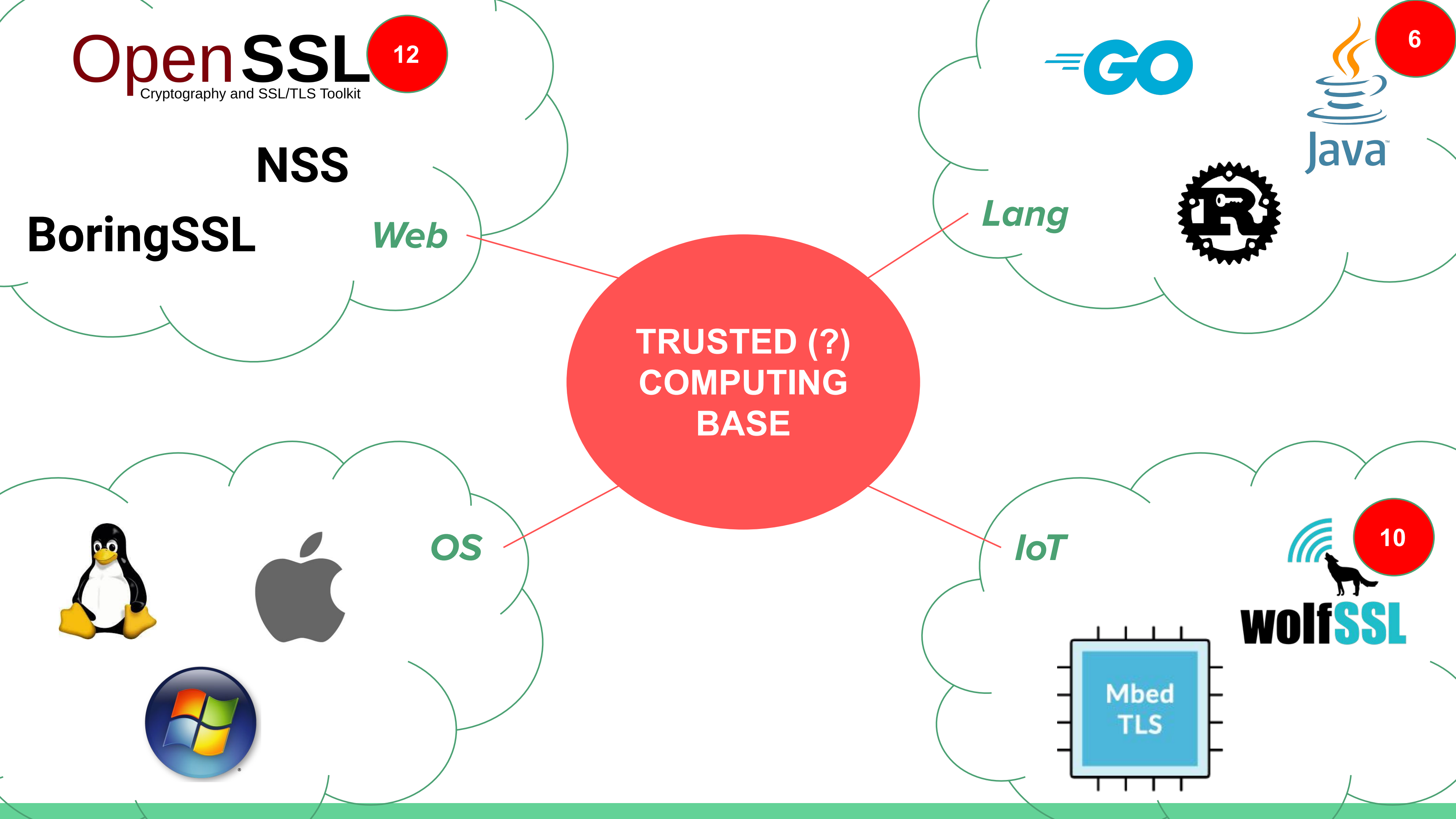
From: On The Insider Security of MLS, Alwen et al. CRYPTO 2022

Verifying Crypto Implementations

Verified Crypto Code	HACL*, Vale, libjade, libcrux, …
Verified Protocol Code	miTLS, LibSignal*, Noise*, MLS*, …



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



OpenSSL
Cryptography and SSL/TLS Toolkit
12
NSS
BoringSSL
Web
GO
Java
6
Lang
TRUSTED (?)
COMPUTING
BASE
OS
IoT
10
wolfSSL
Mbed
TLS

Many Bugs in Classic Crypto Code

CVE-2022-21449: Psychic Signatures in Java

Neil Madden

19 April, 2022

cryptography, Security

API security, cryptography, Java, jose, jwt, web-security

The long-running BBC sci-fi show *Doctor Who* has a recurring plot device where the Doctor manages to get out of trouble by showing an identity card which is actually completely blank. Of course, this being Doctor Who, the card is really made out of a special "psychic paper", which causes the person looking at it to see whatever the Doctor wants them to see: a security pass, a warrant, or whatever.

https://neilmadden.blog/2022/04/19/psychic-signatures-in-java/

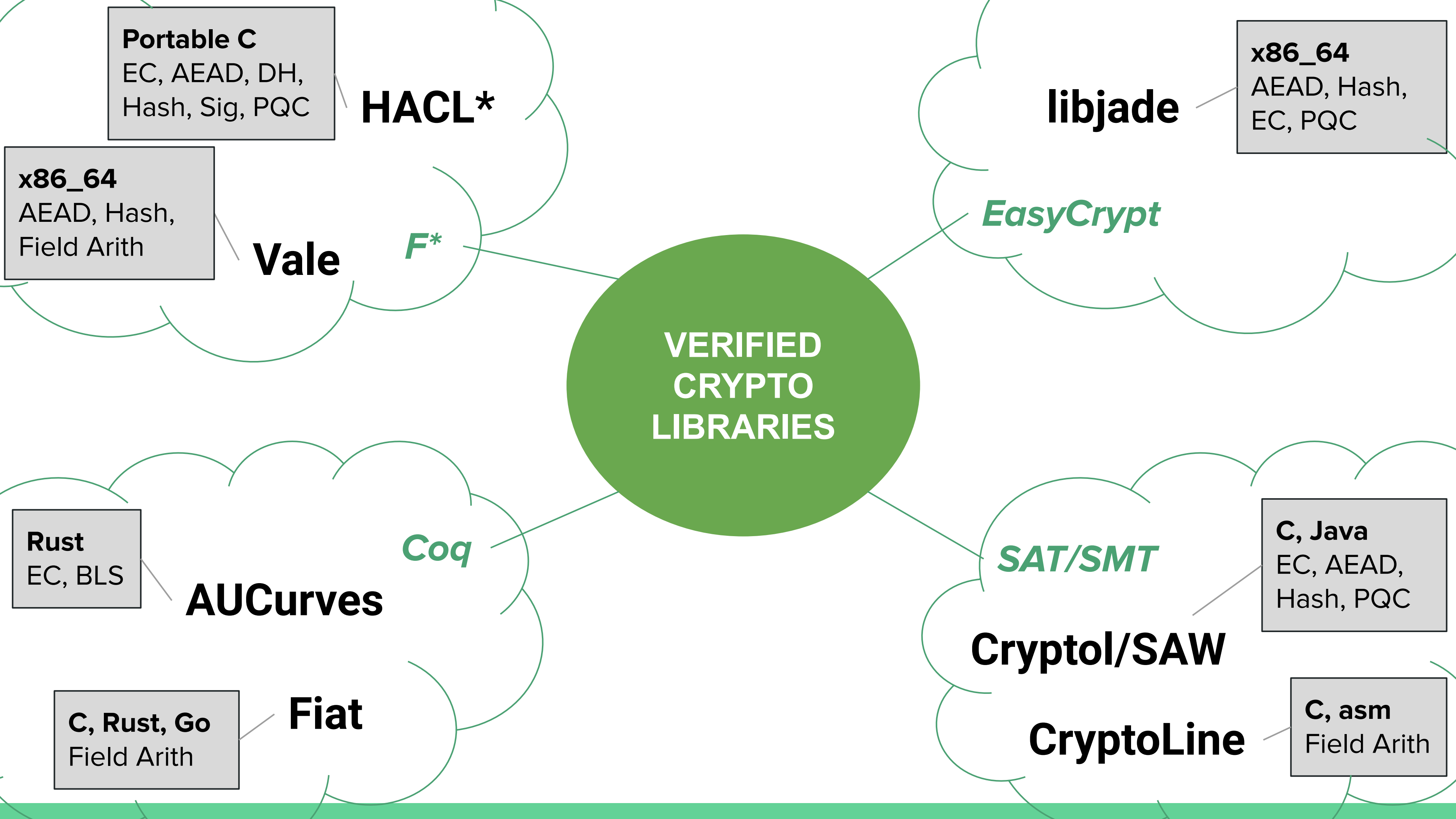
Many Bugs in Classic Crypto Code

This is why the very first check in the ECDSA verification algorithm is to ensure that *r* and *s* are both >= 1.

Guess which check Java forgot?

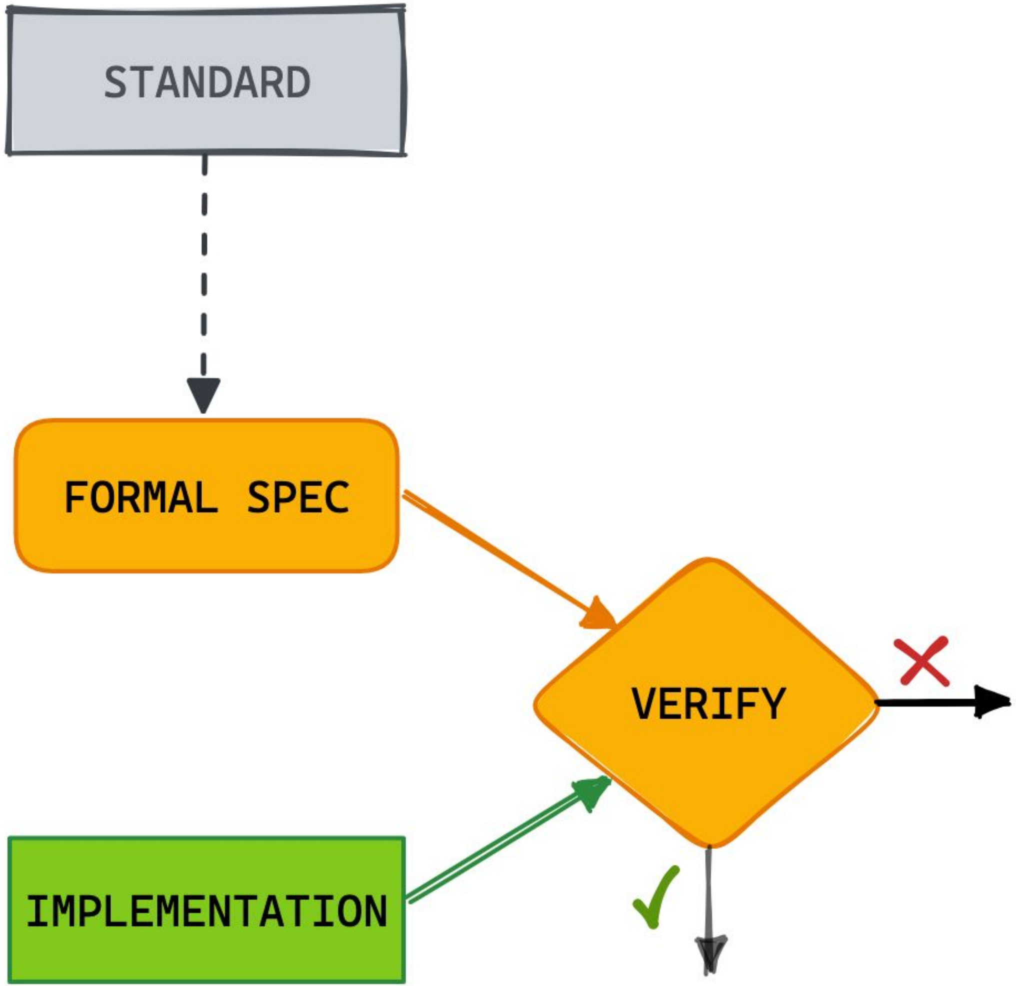
That's right. Java's implementation of ECDSA signature verification didn't check if r or s were zero, so you could produce a signature value in which they are both 0 (appropriately encoded) and Java would accept it as a valid signature for any message and for any public key. The digital equivalent of a blank ID card.

https://neilmadden.blog/2022/04/19/psychic-signatures-in-java/

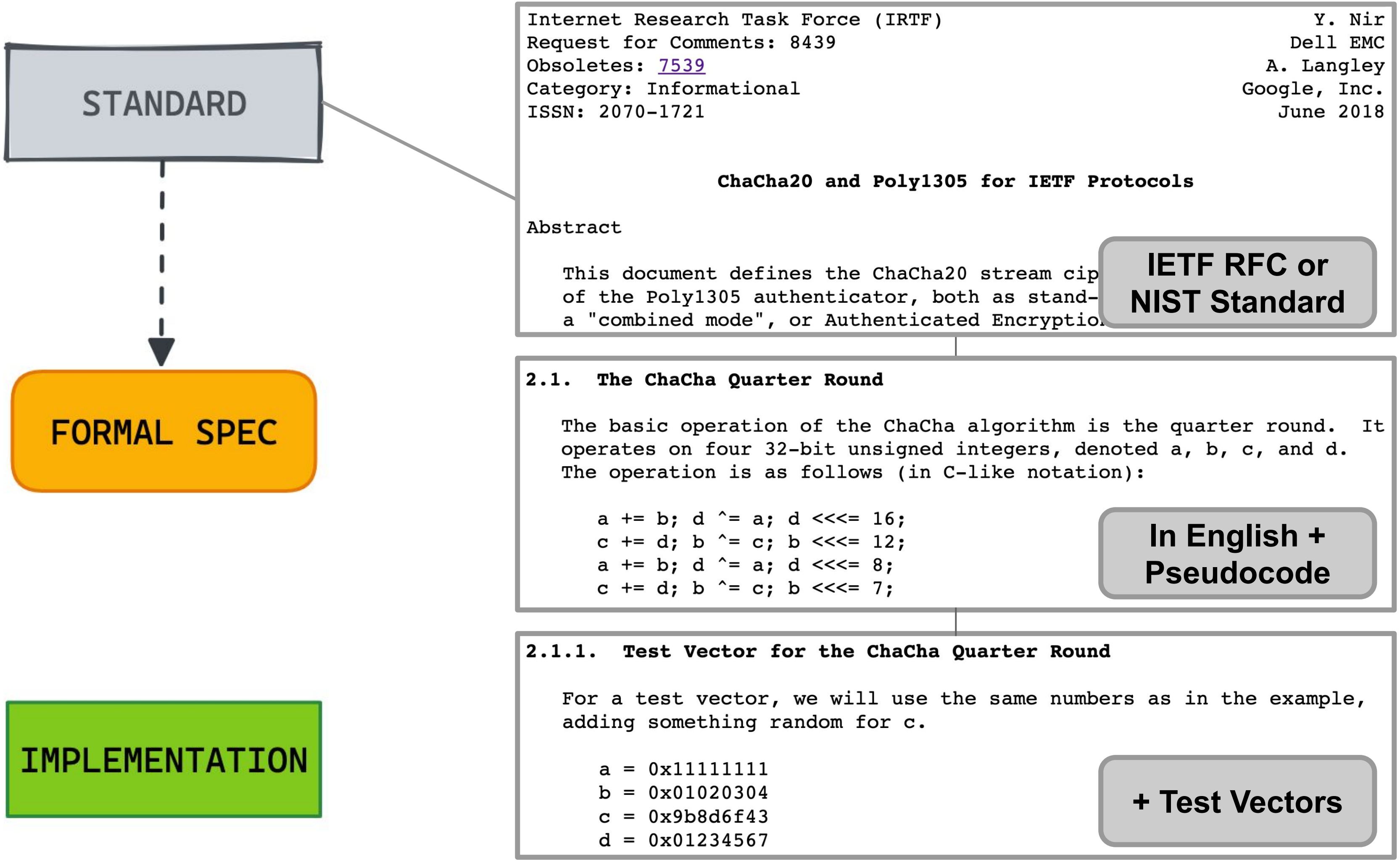

VERIFIED CRYPTO LIBRARIES
HACL*
Portable C
EC, AEAD, DH, Hash, Sig, PQC
Vale
x86_64
AEAD, Hash, Field Arith
F*
libjade
x86_64
AEAD, Hash, EC, PQC
EasyCrypt
Coq
AUCurves
Rust
EC, BLS
Fiat
C, Rust, Go
Field Arith
SAT/SMT
Cryptol/SAW
C, Java
EC, AEAD, Hash, PQC
CryptoLine
C, asm
Field Arith

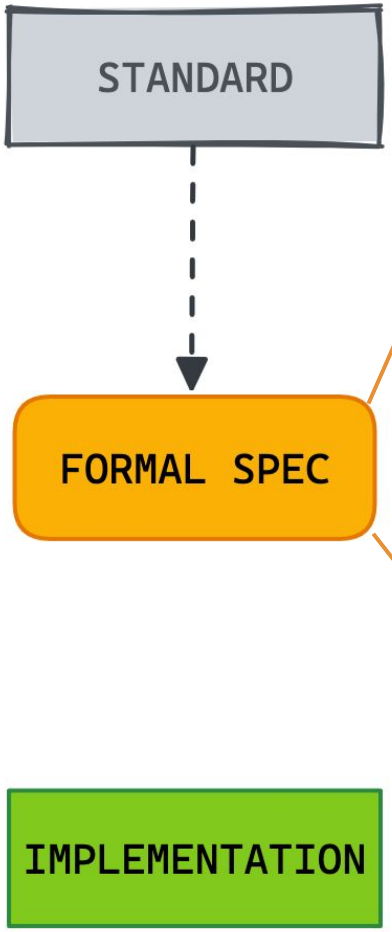
Good news: For any modern crypto algorithm, there is probably a verified implementation.

But... research code with low-level APIs, and specs written in unfamiliar formal languages.



Verified Cryptography Workflow


STANDARD
FORMAL SPEC
IMPLEMENTATION
Internet Research Task Force (IRTF)
Request for Comments: 8439
Obsoletes: 7539
Category: Informational
ISSN: 2070-1721
Y. Nir
Dell EMC
A. Langley
Google, Inc.
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
IETF RFC or NIST Standard
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;
In English + Pseudocode
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
+ Test Vectors

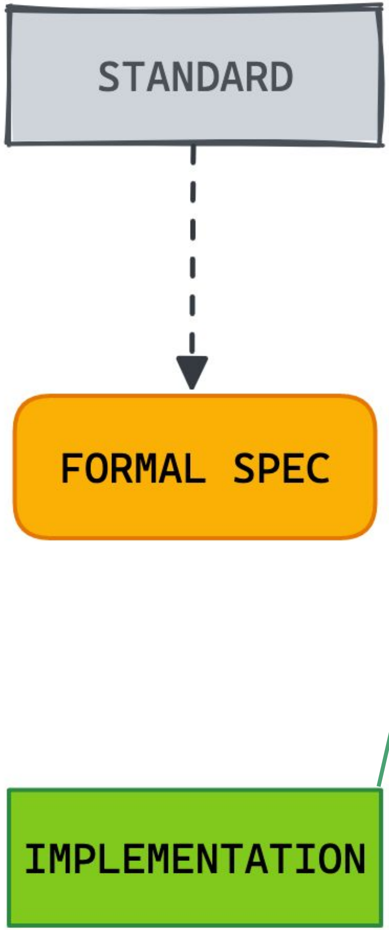



```
let line (a:idx) (b:idx) (d:idx) (s:rotval U32) (m:state) : Tot state =
  let m = m.[a] ← (m.[a] +. m.[b]) in
  let m = m.[d] ← ((m.[d] ^. m.[a]) <<<. s) in m

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)



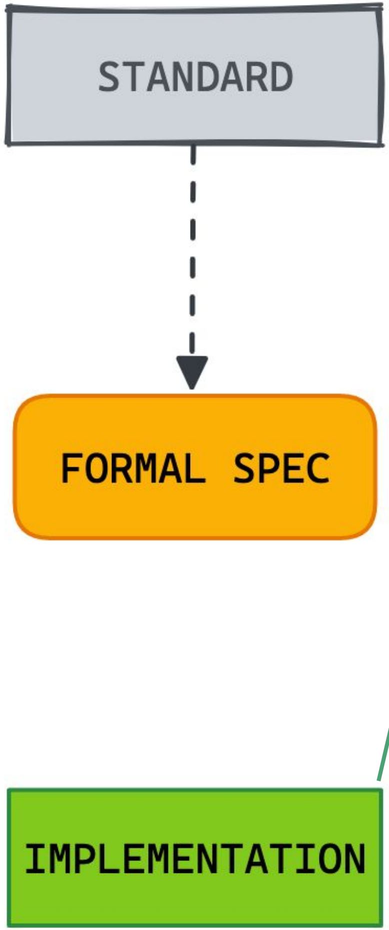
F* Implementation

```
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

```
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

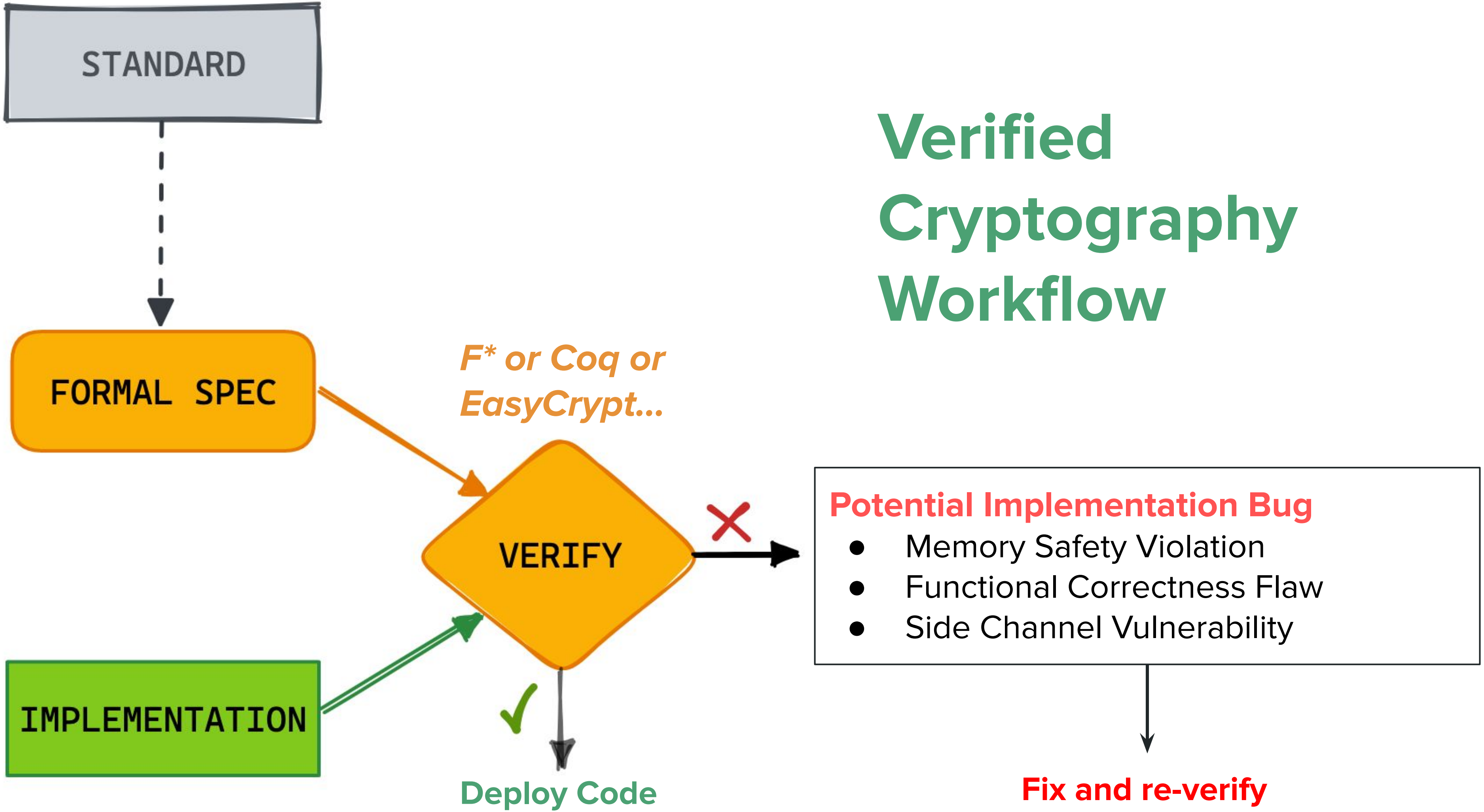
```
inline fn __line_ref(reg u32[16] k,
                     inline int a b c r)
                     -> reg u32[16]
{
  k[a] += k[b];
  k[c] ^= k[a];
  _, _, k[c] = #ROL_32(k[c], r);
  return k;
}

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

But... not always easy to use, extend, or combine code from different 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 pre-conditions

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

```
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" specs

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

```
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)
}
```

ChaCha20 in hacspec

```
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
```

F* 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.
```

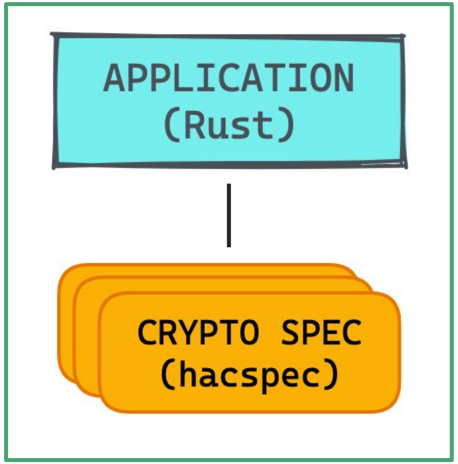
Coq Spec

```
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;
}
```

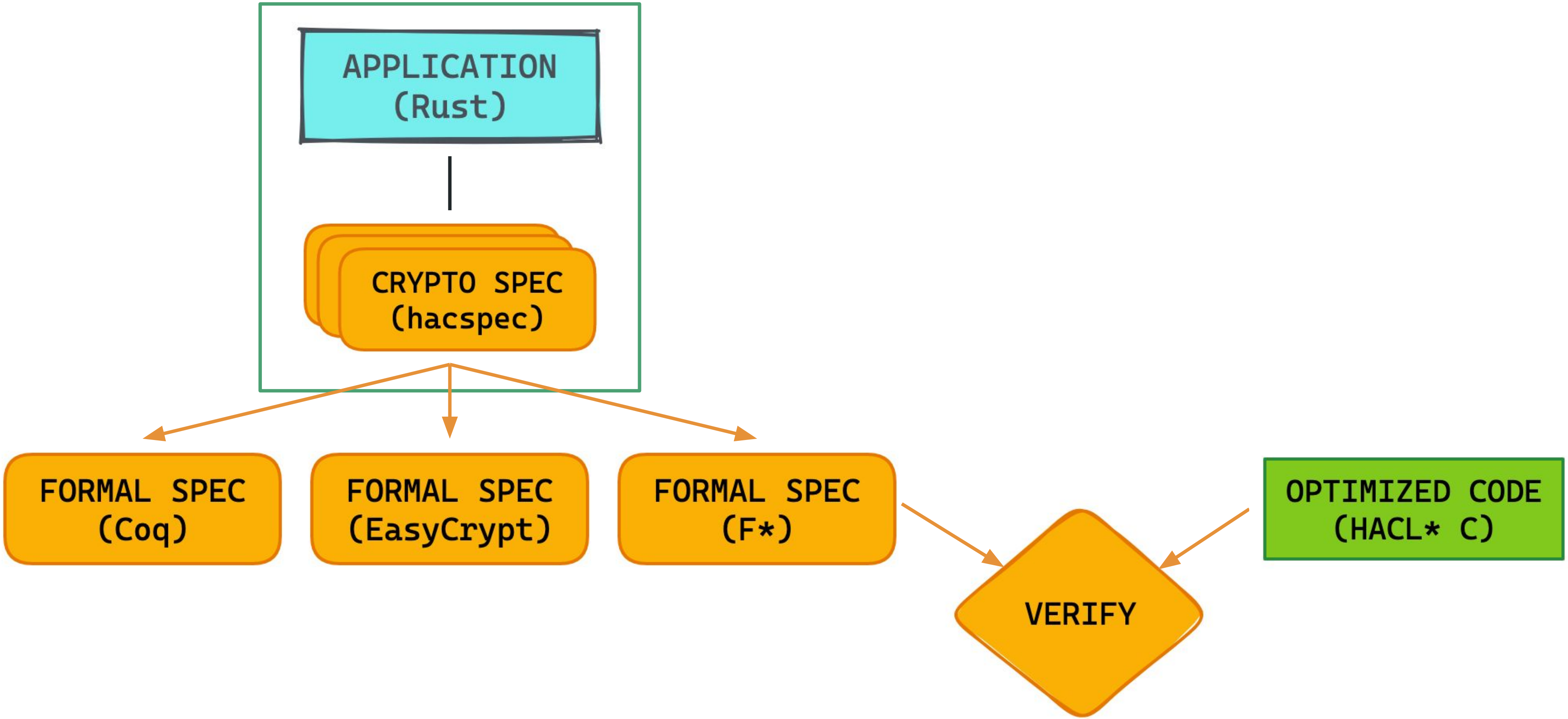
EasyCrypt Spec

Active development: github.com/hacspec

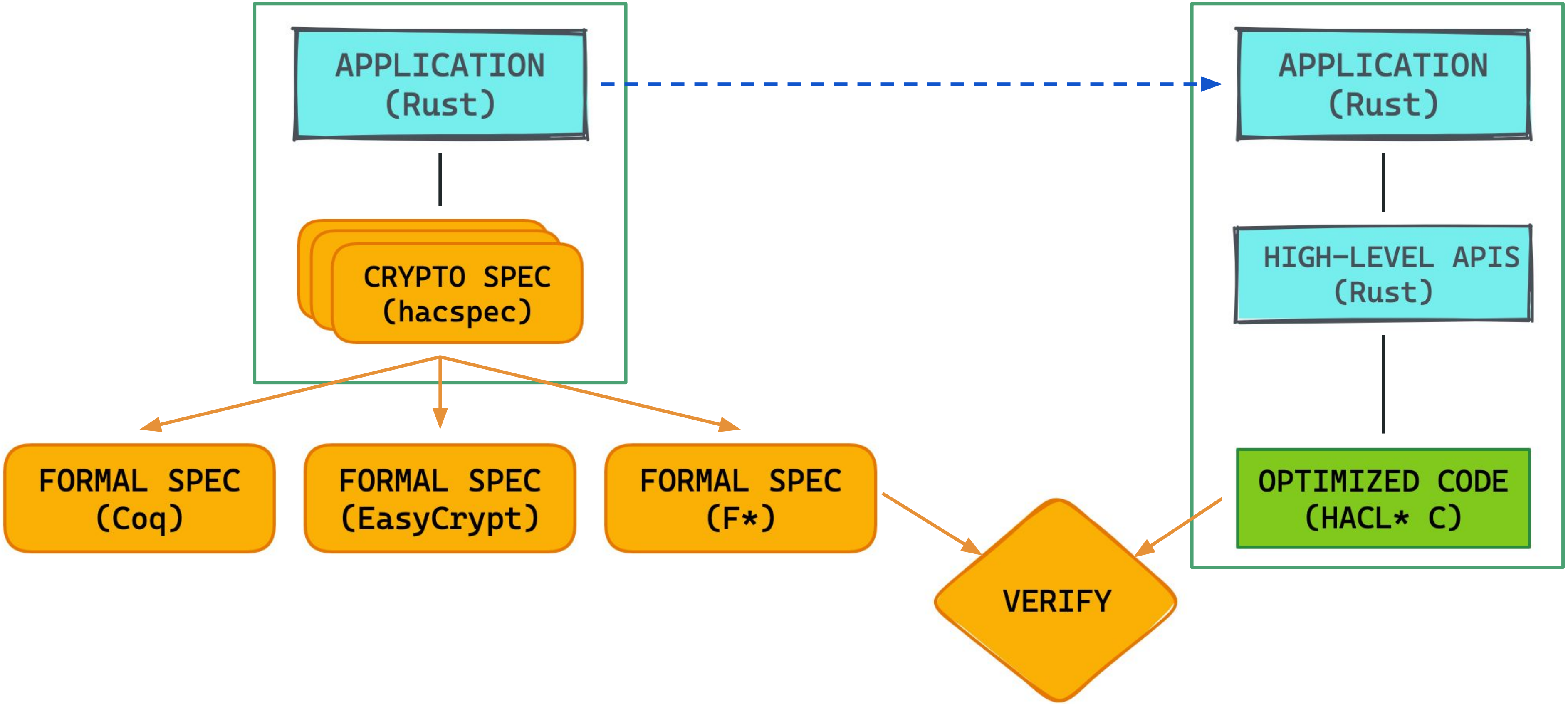
hacspec: towards high-assurance crypto software



hacspec: towards high-assurance crypto software

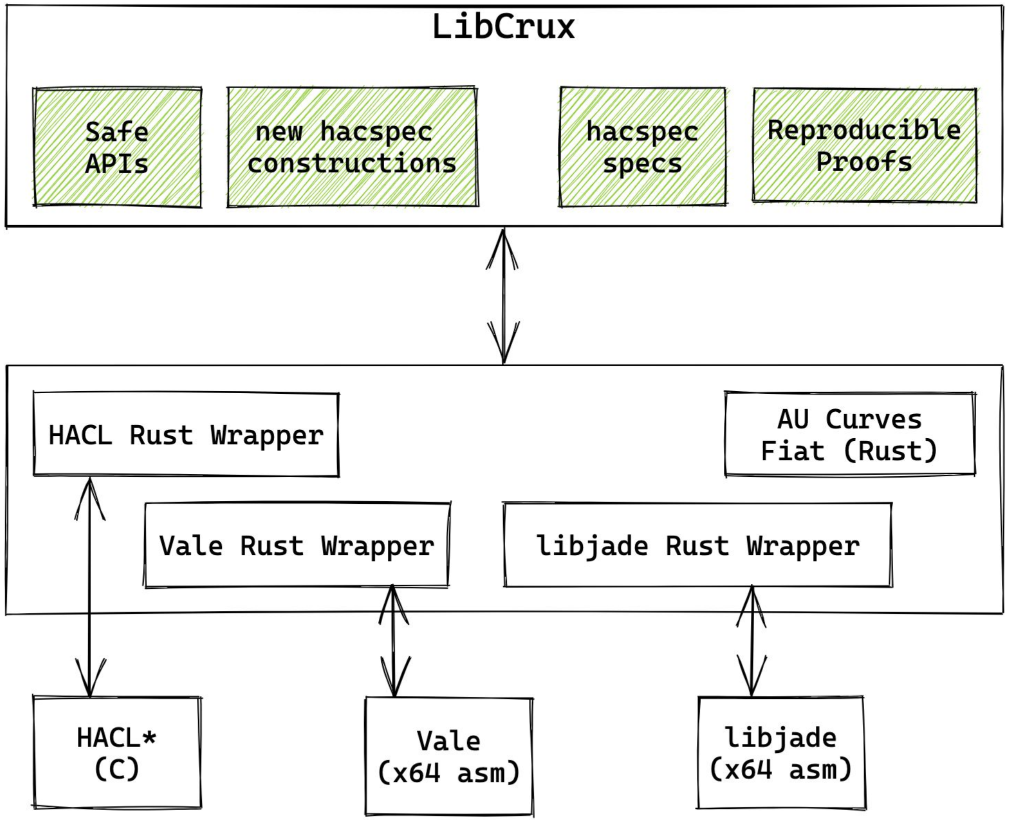


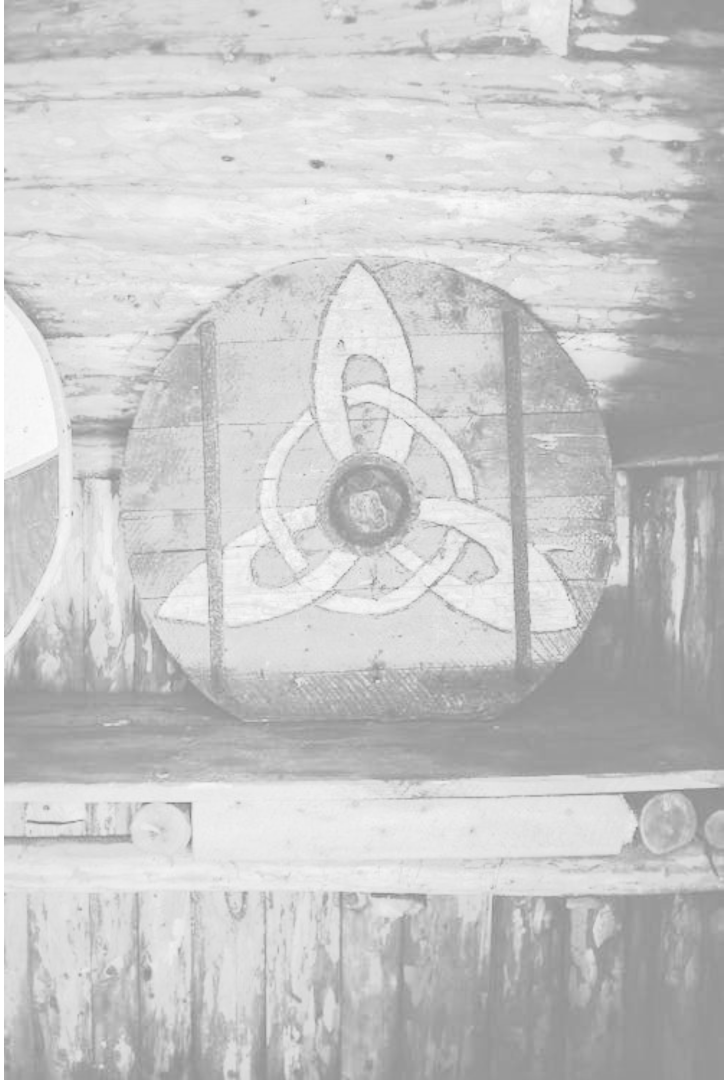
hacspec: towards high-assurance crypto software



libcrux: a library of verified cryptography

libcrux: architecture





Unsafe APIs: Array Constraints

```c
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
);
```

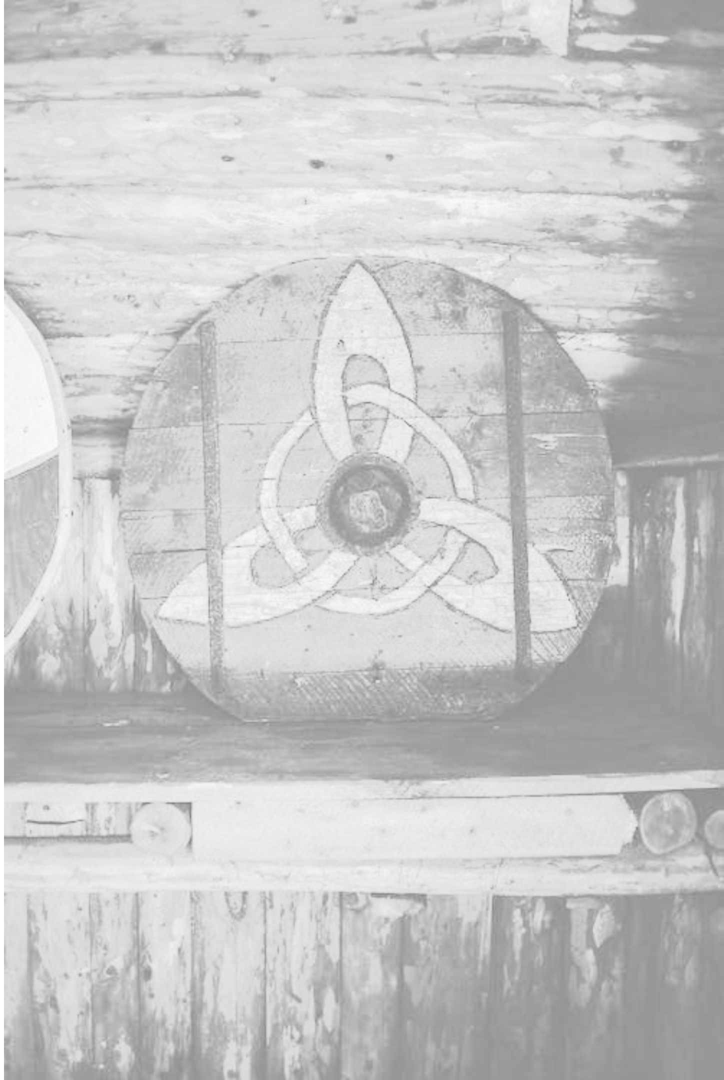
Fixed Length (`*k`, `*n`)

Disjoint (`*cipher`, `*mac`)

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)
```

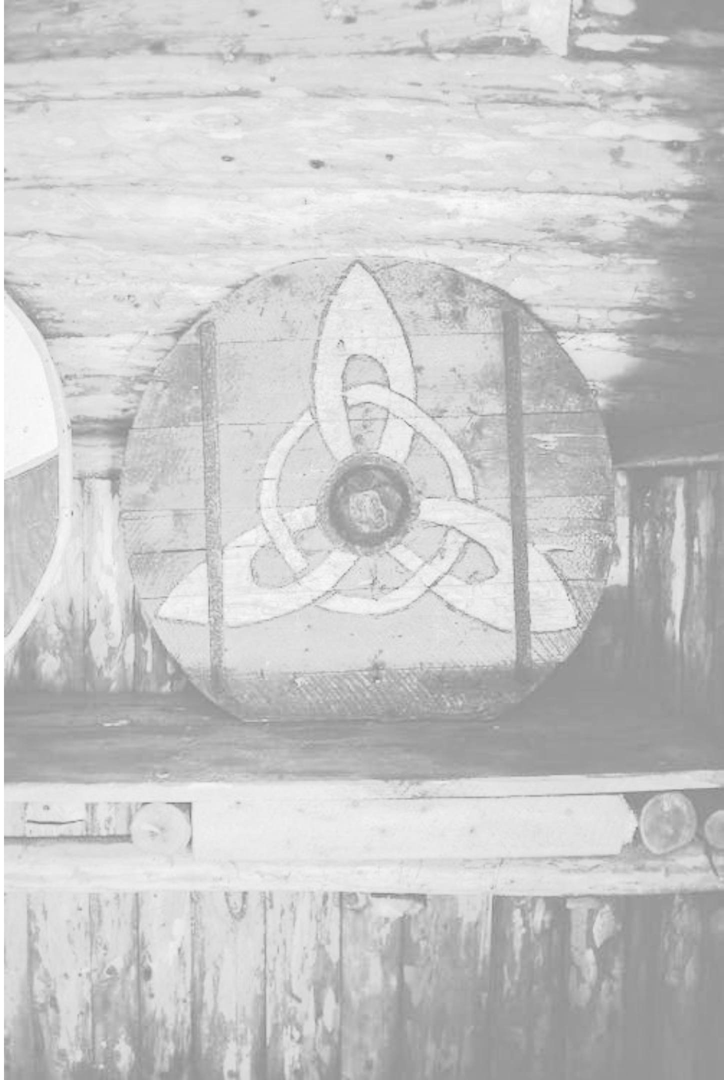
Length Constraints



Verified F* API: Preconditions

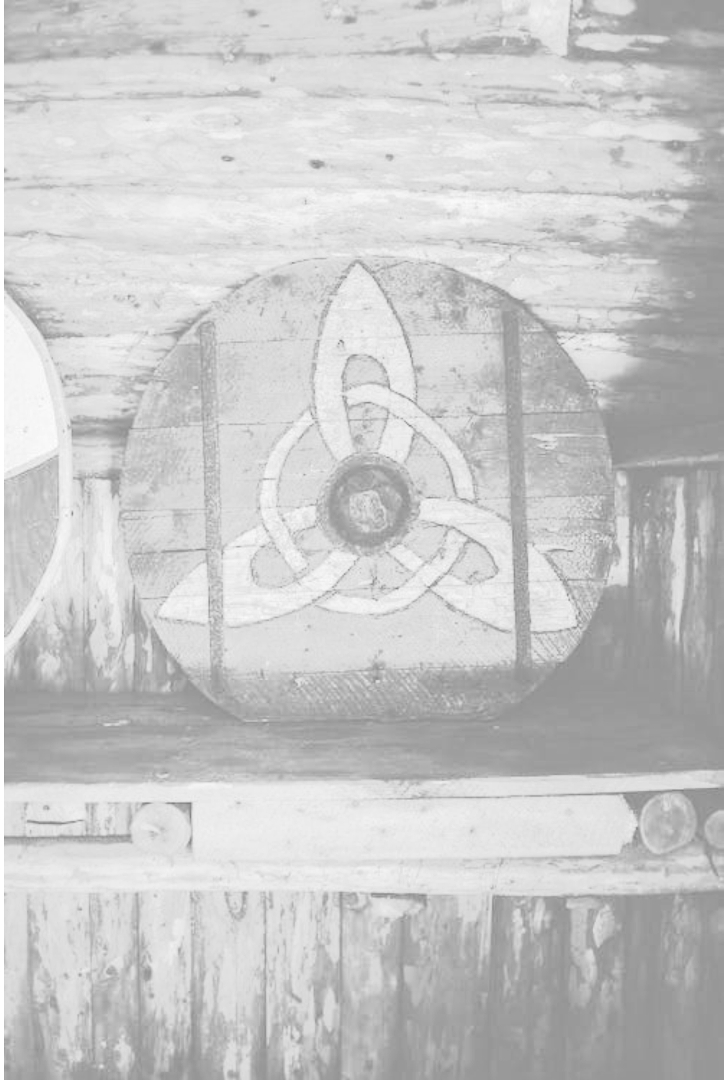
```
let aead_encrypt_st (w:field_spec) =
    key:lbuffer uint8 32ul
  -> nonce:lbuffer uint8 12ul
  -> alen:size_t
  -> aad:lbuffer uint8 alen
  -> len:size_t
  -> input:lbuffer uint8 len
  -> output:lbuffer uint8 len
  -> tag:lbuffer uint8 16ul ->
  Stack unit
  (requires fun h ->
    live h key /\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



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

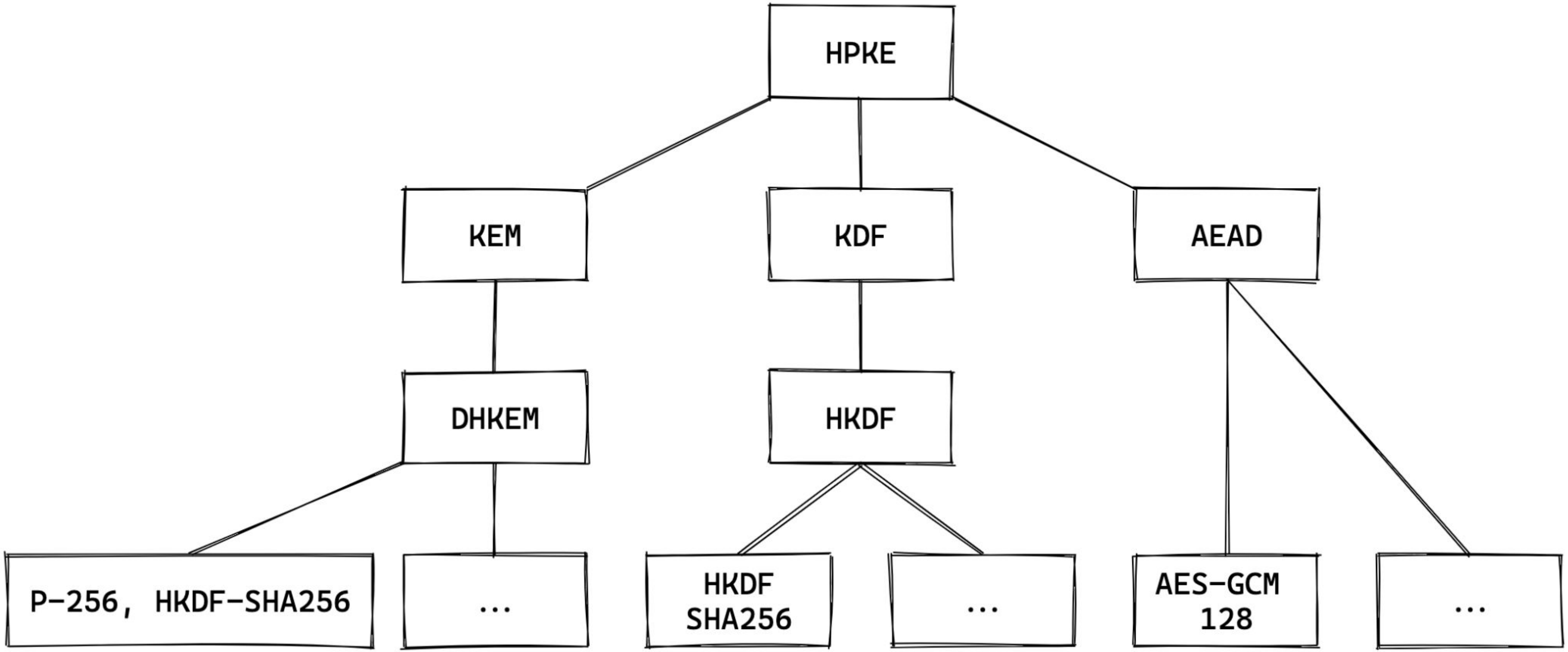
Intel Kaby Lake (ADX, AVX2)

	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)

Building HPKE over libcrux



Ongoing Work: 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

- **Protocol verification** tools are available for analyzing real-world protocols
  - Symbolic analyzers (ProVerif, Tamarin, DY*)
  - Computational provers (CryptoVerif, EasyCrypt, Squirrel, SSProve)
  - Many case studies (HPKE, MLS, TLS 1.3, Noise, Signal)
- **Fast verified code** is available for most modern crypto algorithms
  - Portable C (HACL*, Fiat-Crypto), Assembly (Vale, libjade, CryptoLine)
  - Ongoing work: PQC, ZKP, FHE, MPC, ...
- **hacspec** is a common spec language for multiple verification tools
  - **Try it:** hacspec.org
- **libcrux** provides safe Rust APIs to multiple verified crypto libraries
  - **Try it:** libcrux.org

Thanks!

- HACL*: https://github.com/hacl-star/hacl-star
- Vale: https://github.com/project-everest/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

We are hiring R&D crypto/proof engineers at Inria and Cryspen. Get in touch!

Inria CRYSPEN