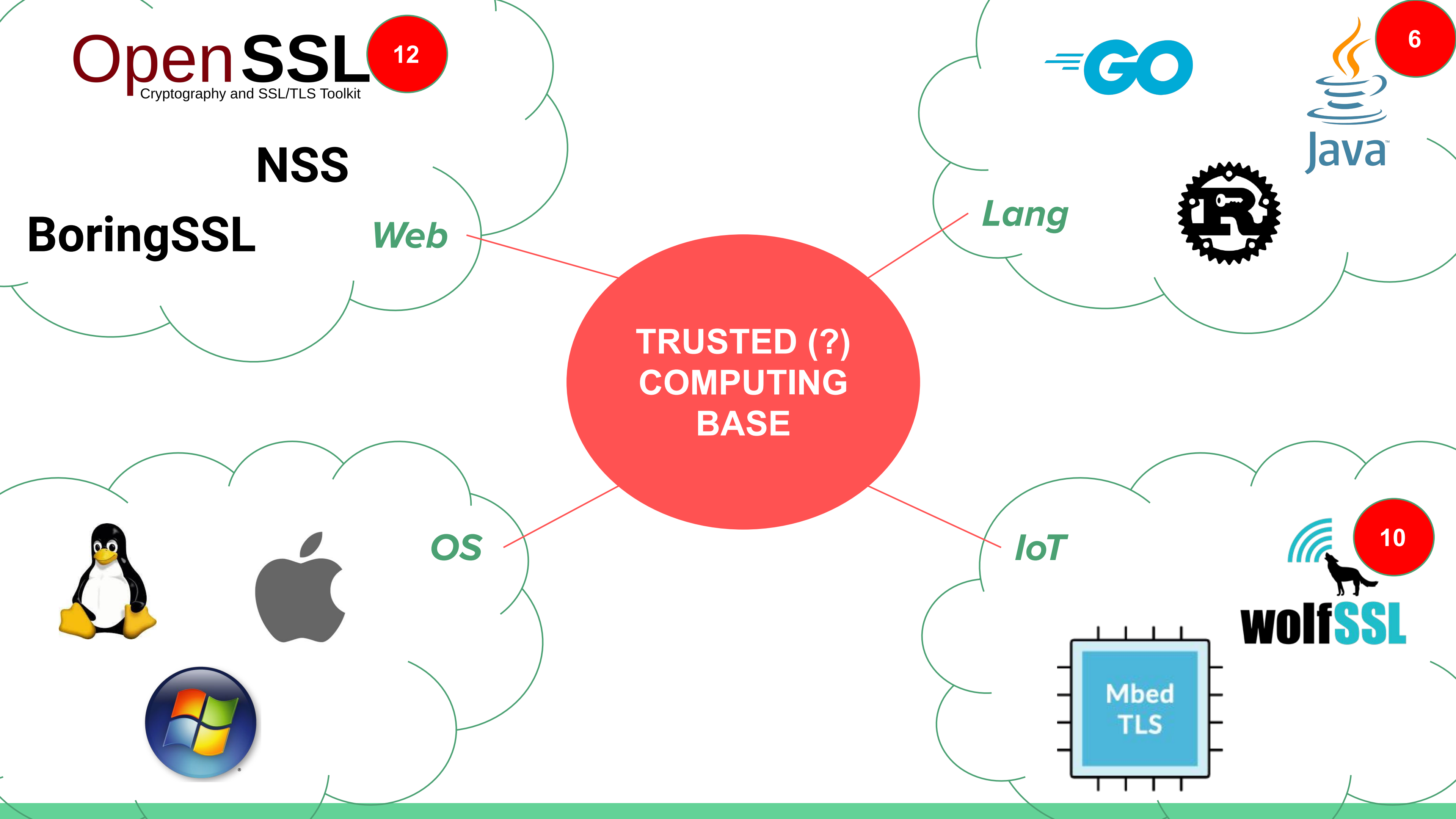
Foundational End-to-End Verification of High-Speed Cryptography

Philipp G. Haselwarter, Aarhus Uni
Benjamin Salling Hvass, AU
Lasse Letager Hansen, AU
Théo Winterhalter, Inria
Catalin Hritcu, MPI
Bas Spitters, AU

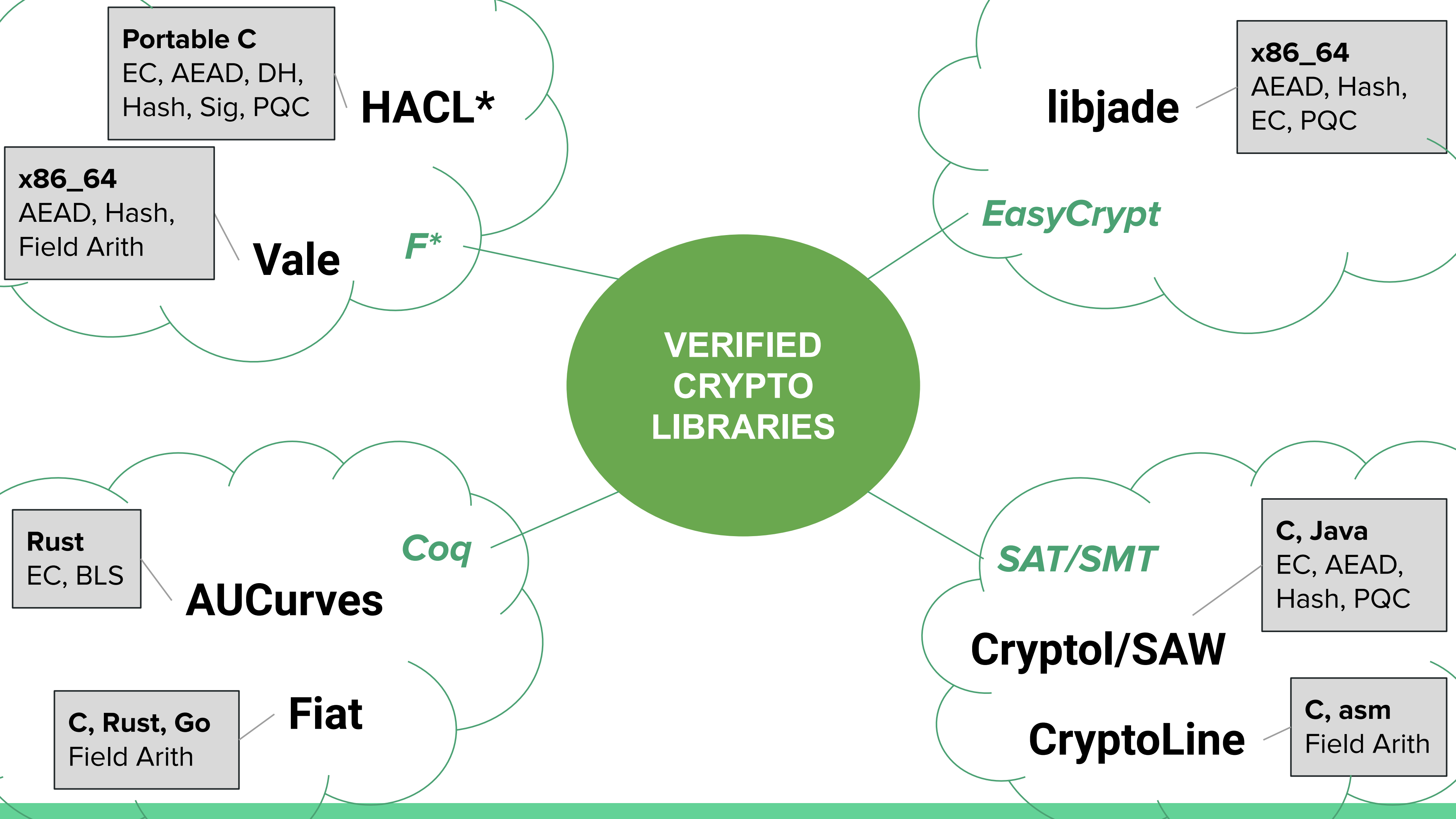
hacspec

a gateway to high-assurance cryptography (RWC)

Franziskus Kiefer, Karthikeyan Bhargavan
Lucas Franceschino, Denis Merigoux
Bas Spitters, Lasse Letager Hansen
Manuel Barbosa, Pierre-Yves Strub



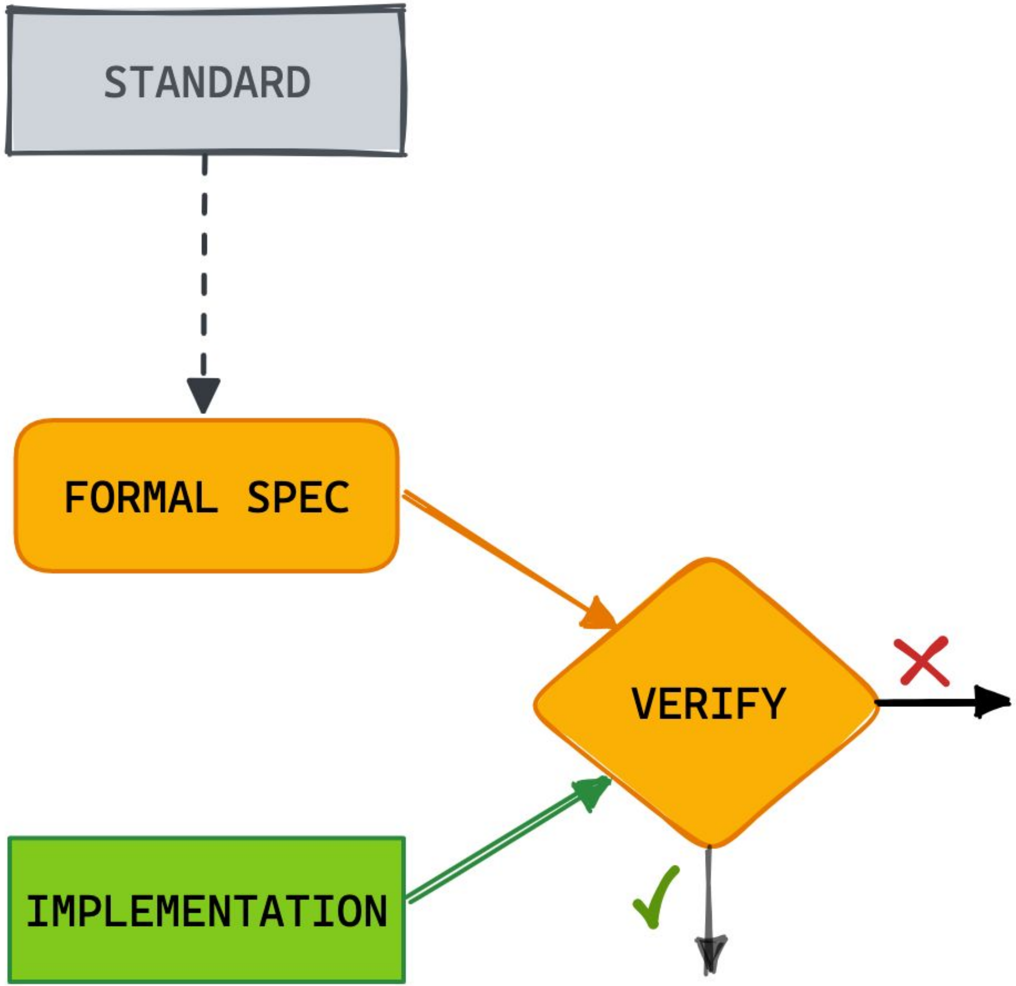
OpenSSL
Cryptography and SSL/TLS Toolkit
12
NSS
BoringSSL
Web
Lang
GO
Java
6
TRUSTED (?)
COMPUTING
BASE
OS
IoT
10
wolfSSL
Mbed
TLS

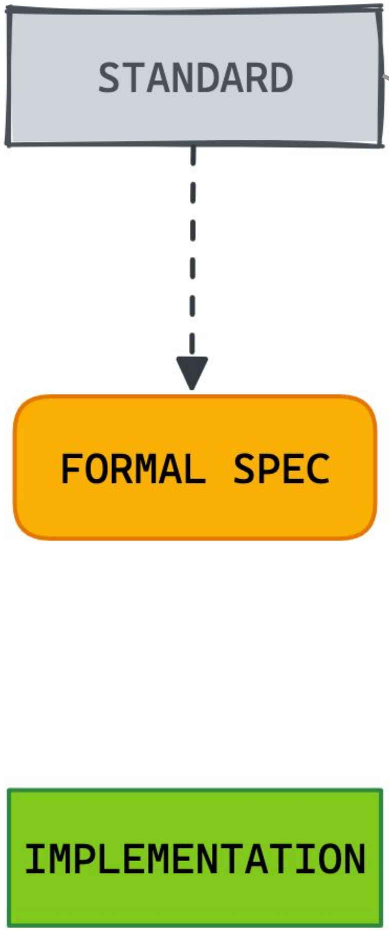

VERIFIED CRYPTO LIBRARIES
F*
HACL*
Portable C
EC, AEAD, DH, Hash, Sig, PQC
Vale
x86_64
AEAD, Hash, Field Arith
EasyCrypt
libjade
x86_64
AEAD, Hash, EC, PQC
Coq
AUCurves
Rust
EC, BLS
Fiat
C, Rust, Go
Field Arith
SAT/SMT
Cryptol/SAW
C, Java
EC, AEAD, Hash, PQC
CryptoLine
C, asm
Field Arith

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

But... specs written in unfamiliar languages.

Verified Cryptography Workflow





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;
```

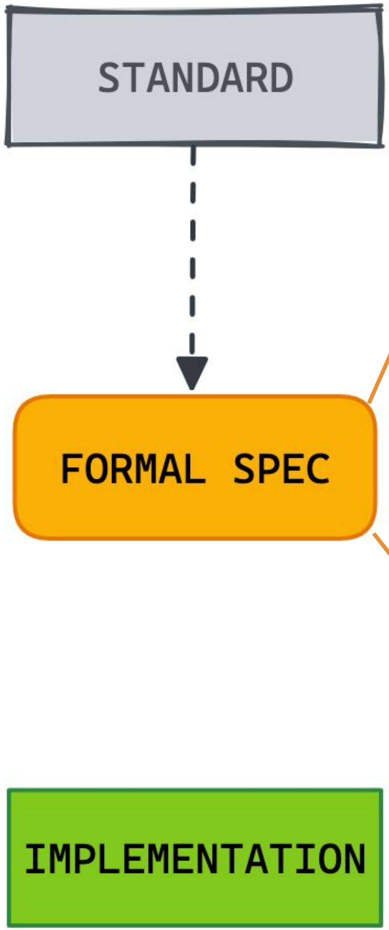
In English + Pseudocode

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

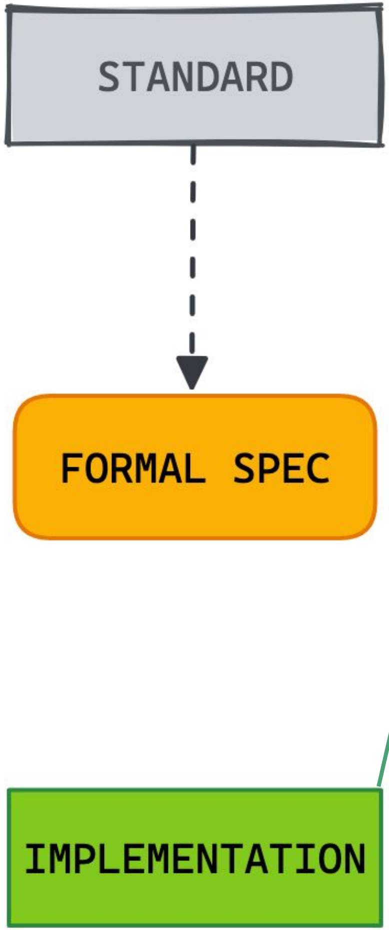



```
let line (a:idx) (b:idx) (d:idx) (s:rotval U32) (m:state) : Tot state =
  let m = m.[a] ← (m.[a] +. m.[b]) in
  let m = m.[d] ← ((m.[d] ^. m.[a]) <<<. s) in m

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)



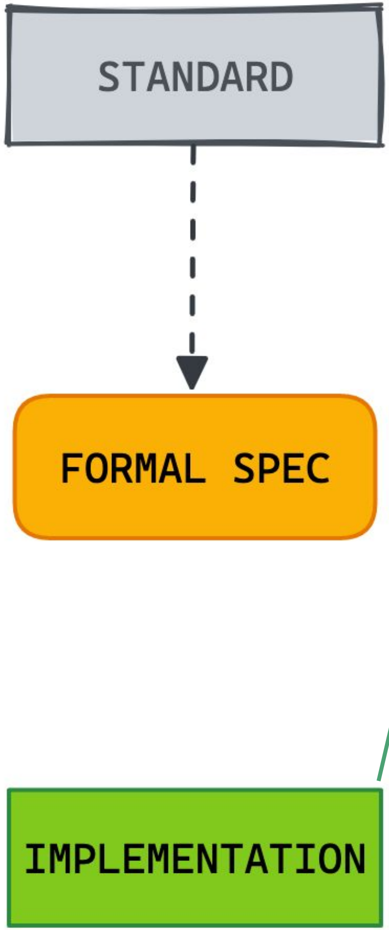
F* Implementation

```
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




```
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;
}
```

Jasmin Implementation

Translate

```
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
...
```

Intel AVX2 Optimized Assembly

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

Ready to use today:

- 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 easy to use, or review, or extend, or combine different verified implementations

- You need to carefully audit the formal specs, written in tool-specific spec languages like F*, Coq, EasyCrypt
- You 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 Tools 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: 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

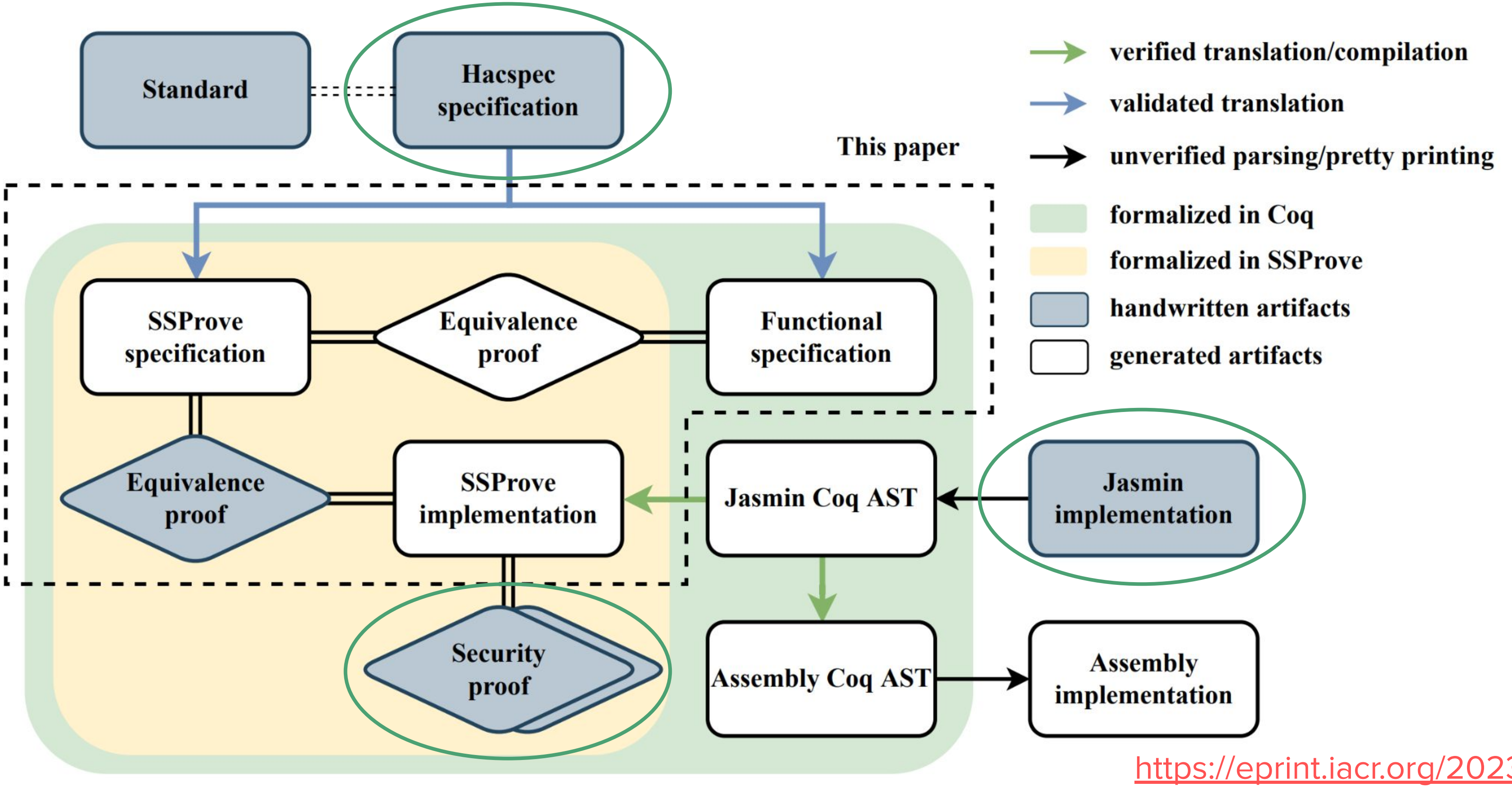
libcrux: a library of verified cryptography

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

Conclusions (libcrux)

- **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 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

The Last Yard: linking hacspec to security proofs



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

Coq

Coq: proof assistant based on dependent type theory

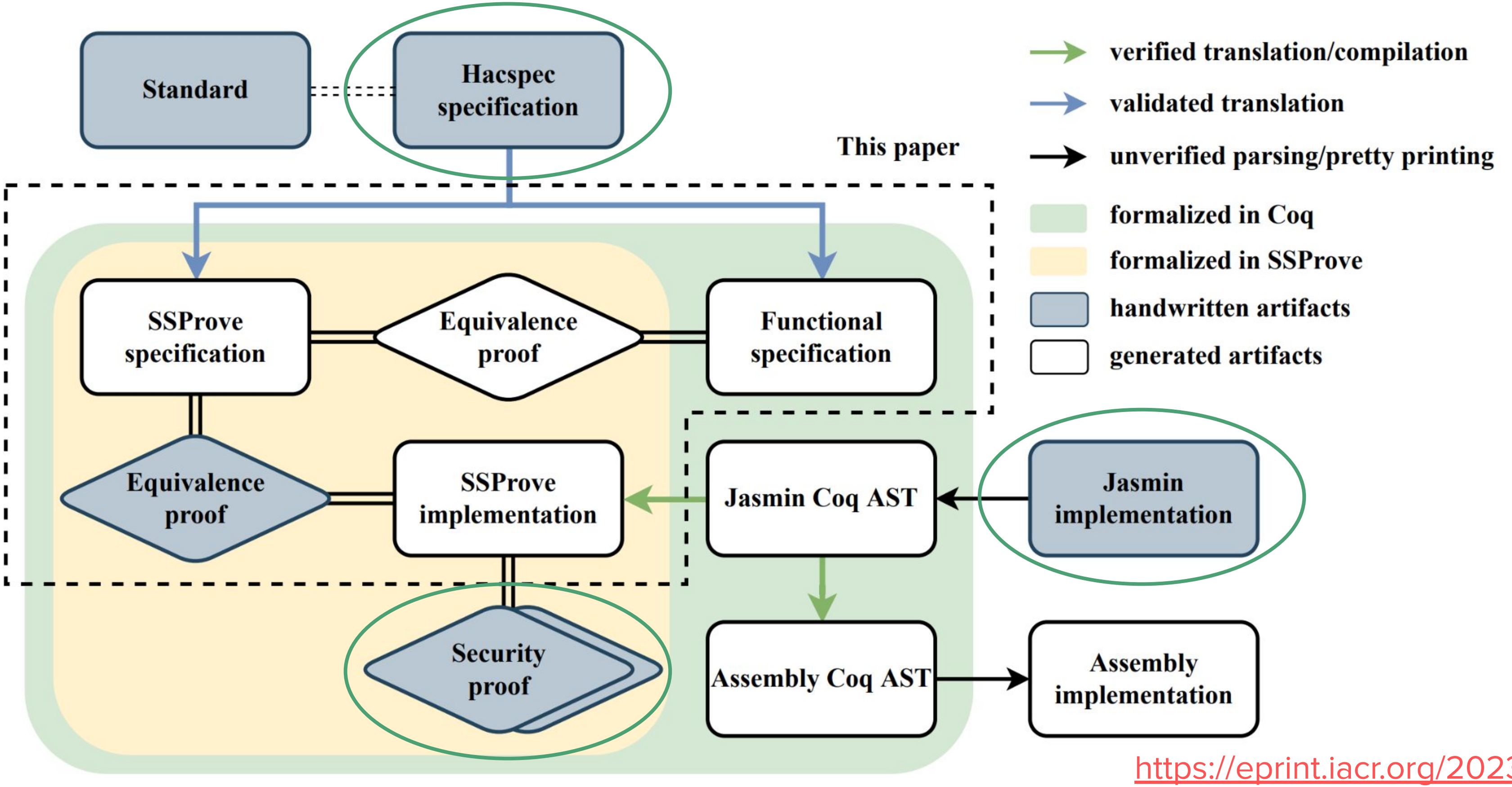
Foundational: all proofs are reduced to a small kernel

Embedded (ocaml-like) functional programming language

Biggest library of formal proofs

Many uses programming language verification

The Last Yard: linking hacspec to security proofs



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

Jasmin

Problem: C-compilers have bugs, cannot be trusted to preserve constant-time

Jasmin language: structured control flow with assembly instructions

Coq verified compiler produces efficient code for x86 and ARM

Compiler does not introduce timing side-channel attacks

https://github.com/jasmin-lang/jasmin/wiki

Hacspec and jasmin

Small imperative language **L** embedded in Coq

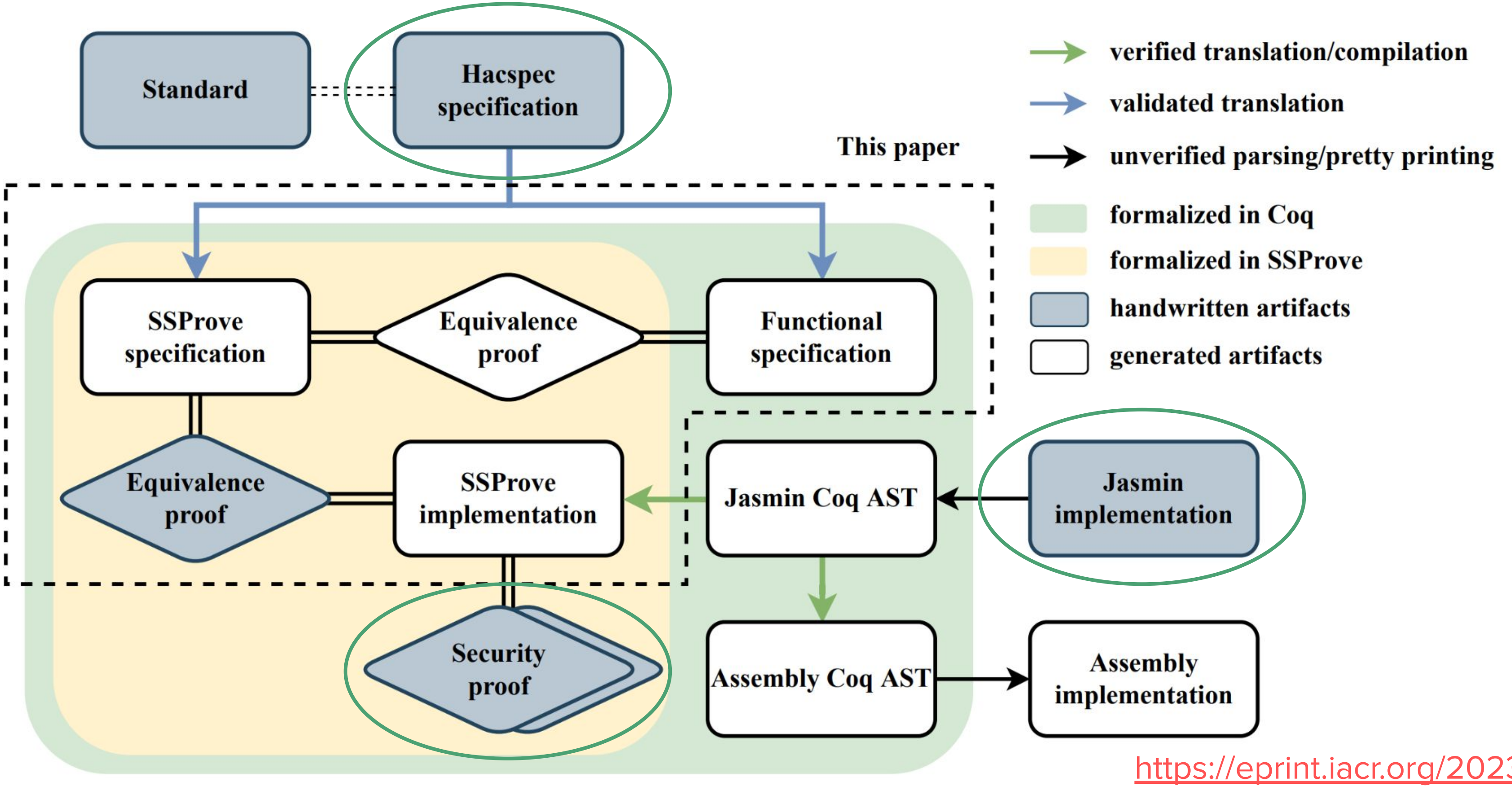
We connected the functional interpretation of a hacspec program with an imperative interpretation

Automatic modular equivalence proofs

+ equivalence proofs with embedded jasmin AST

Framework for functional correctness of jasmin wrt hacspec

The Last Yard: linking hacspec to security proofs



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

Cryptographic security

Computational model of security (game hopping)

Dedicated tool support: Easycrypt

Not connected to huge mathematical libraries, not foundational

SSProve library in Coq

Build on math-comp mathematical library, includes game hopping, categorical semantics.

State **S**eparating **P**roofs: modular proof technique, similar to Joy of Cryptography

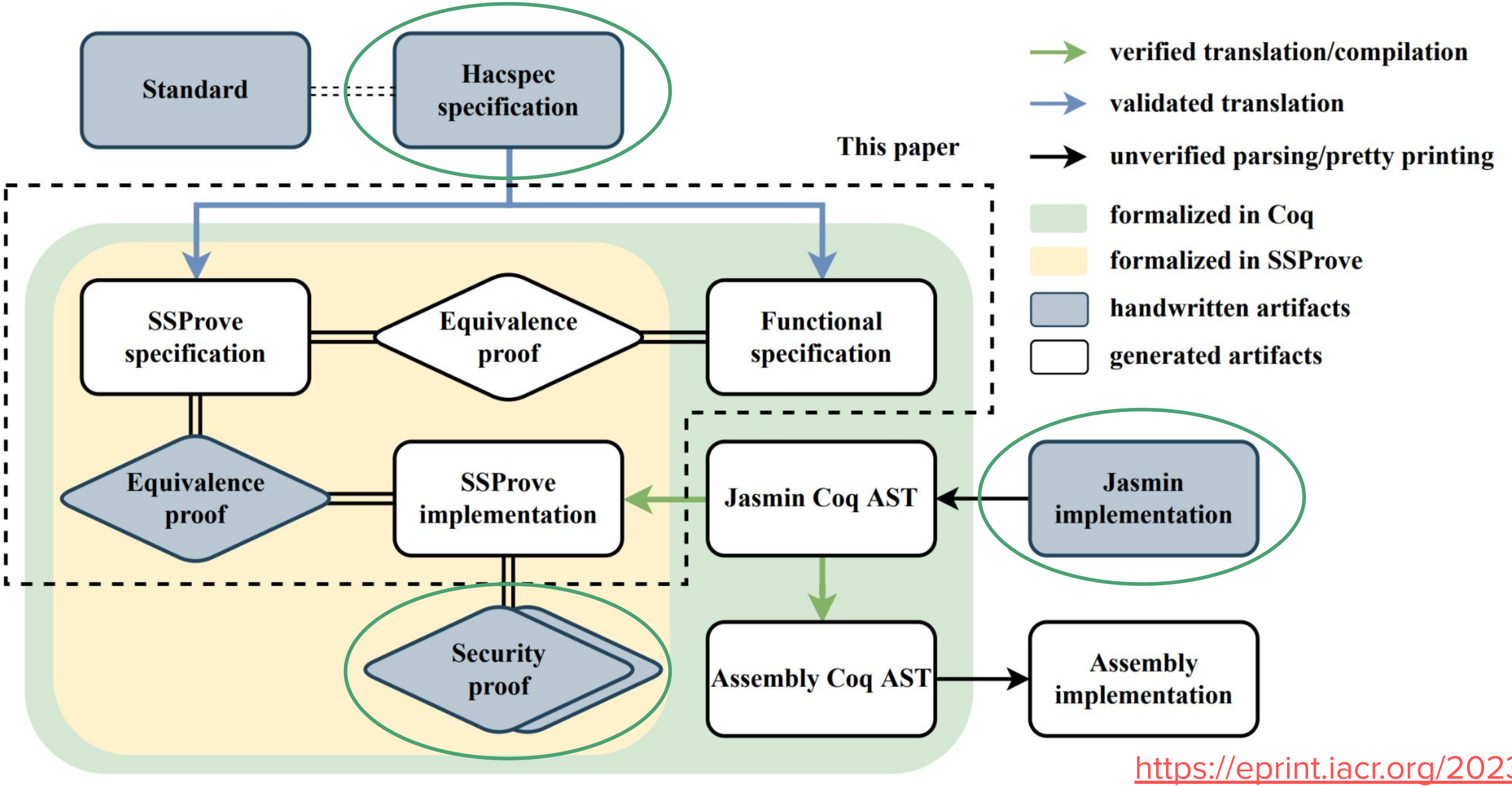
AES is cryptographically secure

Case study:

existing AES jasmin implementation is cryptographically secure

Ciphertext indistinguishability (IND-CPA)

The Last Yard: linking hacspec to security proofs



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

Coq Verified pipeline from:

- specification (hacspec) to
- efficient implementation (jasmin)
- verified correctness (Coq)

Specifically:

- AES in hacspec
- with existing jasmin implementation
- IND-CPA security in SSProve