Secure Software with Formal Guarantees

Using Hax

CRYSPEN


Kiel
Stralsund
Rostock
Lübeck
Świnoujście
Great Britain
Manchester
Liverpool
Sheffield
Groningen
Bremen
Szczecin
Gorzów Wielkopolski
Norwich
Birmingham
Amsterdam
Hanover
Potsdam
Magdeburg
Cambridge
Bielefeld
WALES
ENGLAND
Oxford
Netherlands
Rotterdam
Bristol
London
Dortmund
Cardiff
Essen
Kassel
Leipzig
Antwerp
Düsseldorf
Dresden
Brussels
Germany
Exeter
Bournemouth
Cologne
Lille
Liège
Plymouth
Belgium
Frankfurt
Prague
English Channel
Pilsen
Luxembourg
Mannheim
Czechia
Nuremberg
Rouen
Guernsey
Caen
Jersey
Karlsruhe
Regensburg
Stuttgart
Versailles
Saint-Malo
Strasbourg
Brest
Augsburg
Linz
Munich
Rennes
Le Mans
Quimper
Freiburg im Breisgau
Lorient
Salzburg
Angers
Tours
Dijon
Zürich
Austria
Nantes
Basel
Graz

Inria

cyberagentur

CAMPUS CYBER

ZIM
Zentrales
Innovationsprogramm
Mittelstand

erc
European Research Council
Established by the European Commission

Bundesministerium
für Bildung
und Forschung

XMTP

SANDBOXAQ

BWI
IT für Deutschland

Google

NYM

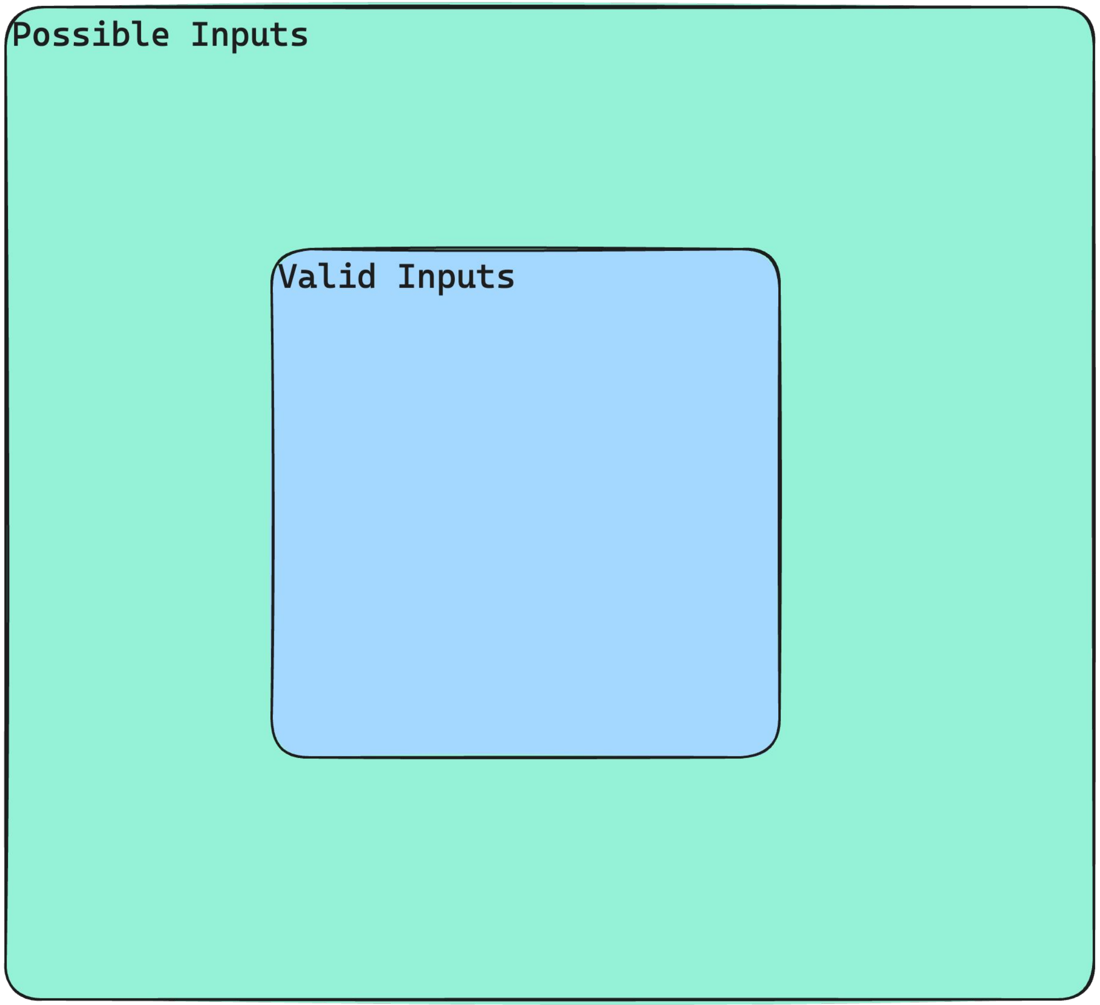
CISCO™

moz://a

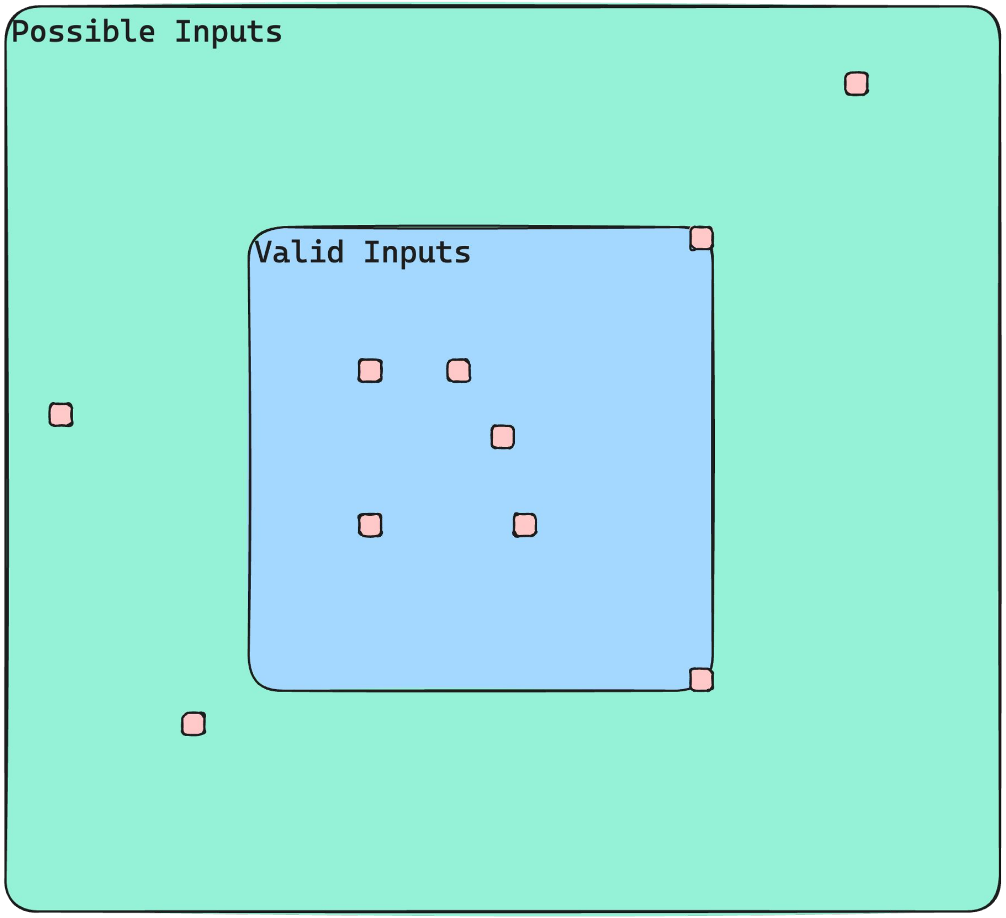
Building Secure Software

[...] testing is a necessary but insufficient step in the development process to fully reduce vulnerabilities at scale [...]

THE WHITE HOUSE
WASHINGTON


Possible Inputs
Valid Inputs

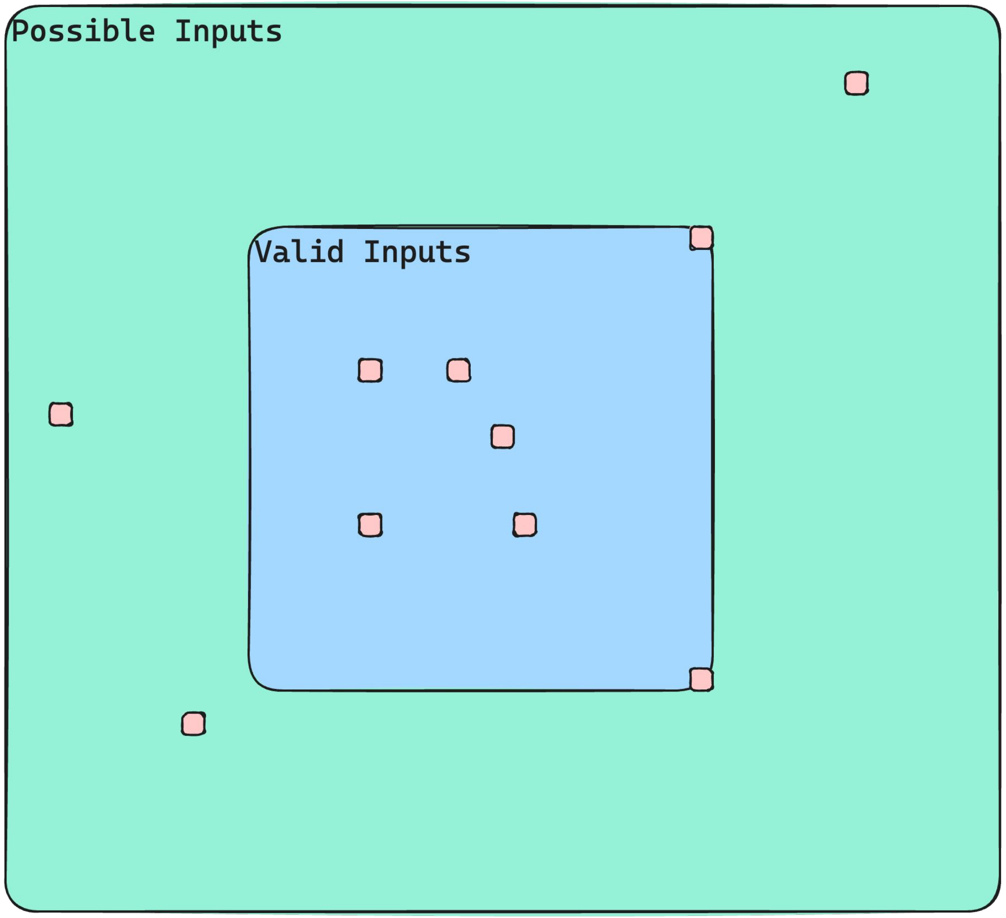
Testing



Testing

Wycheproof
ECDSA P256

471 Tests

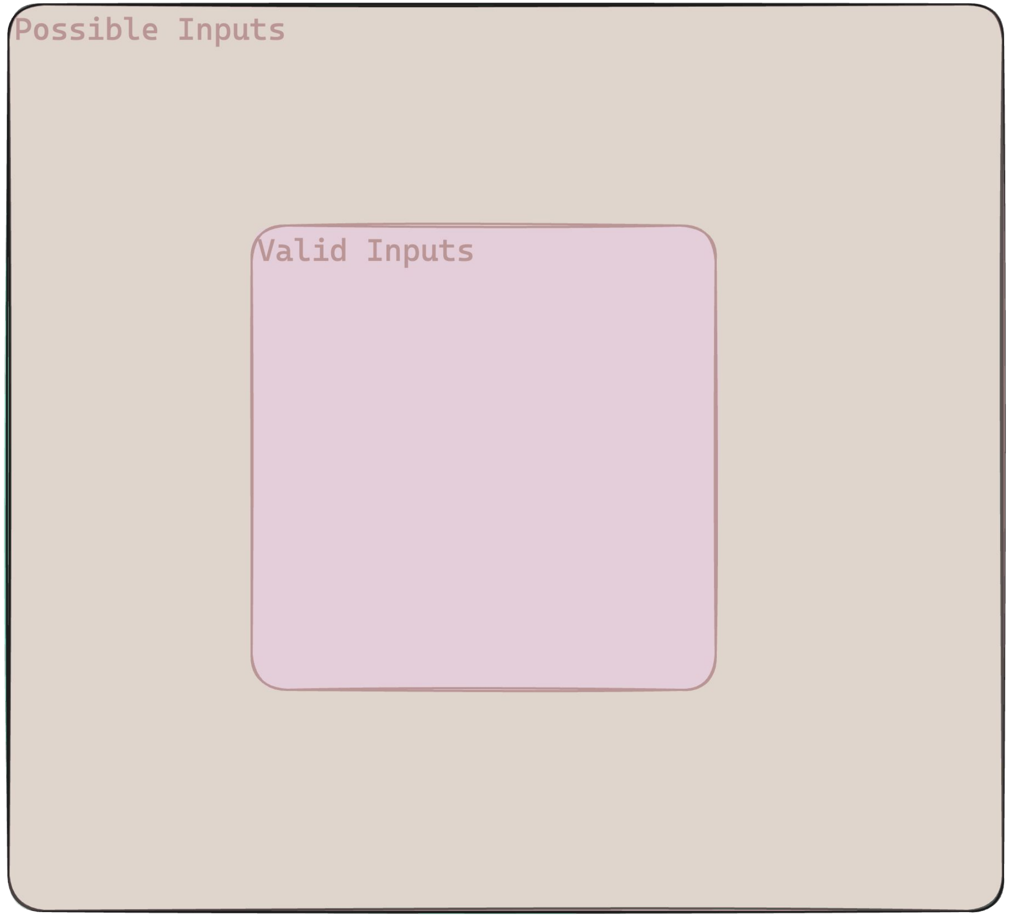


Possible
Inputs

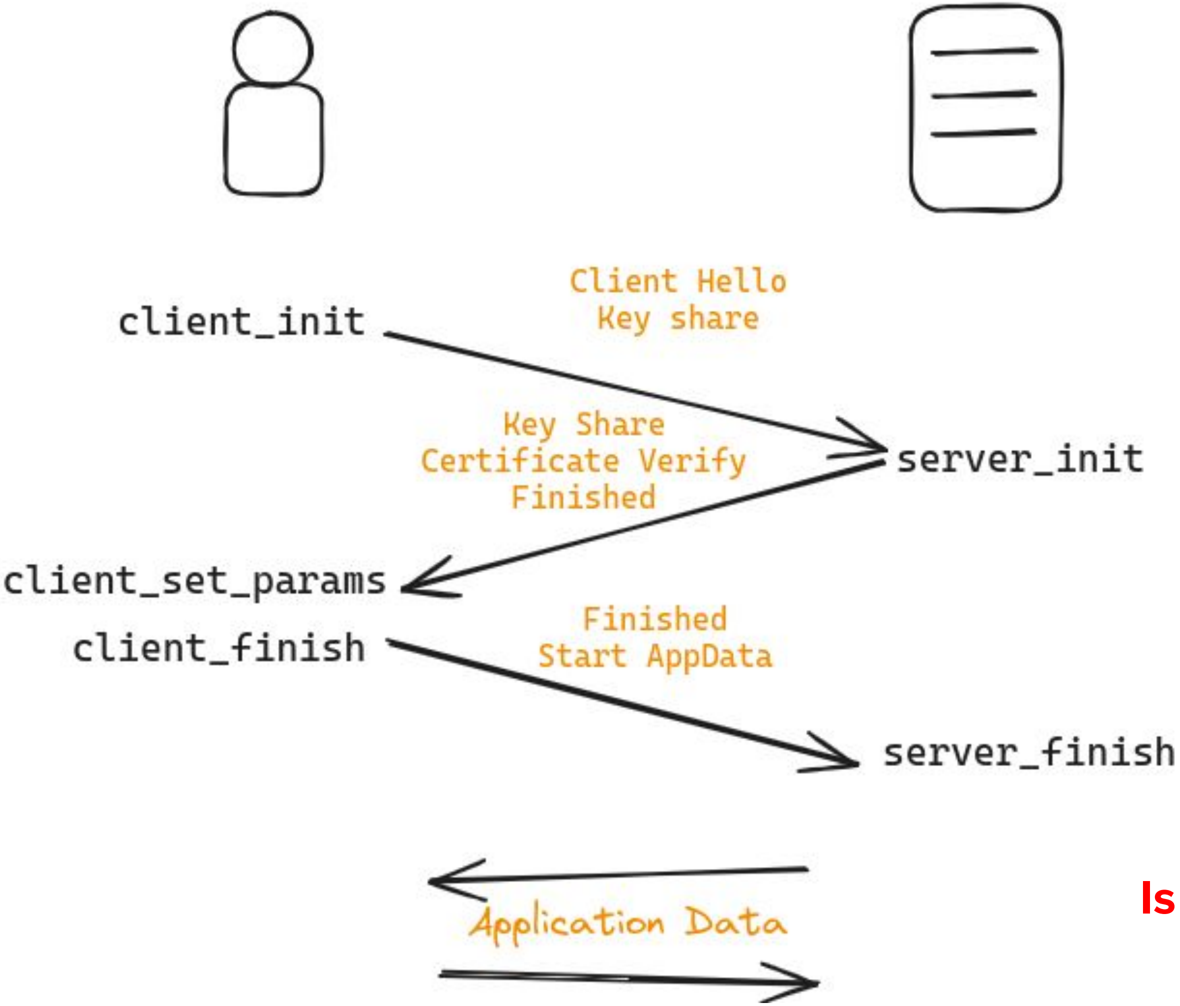
64 bytes
Signature

64 bytes
public key

Verification



TLS



Is this secure?

"[...] correctness is defined as the ability of a piece of software to meet a specific [...] requirement"

THE WHITE HOUSE
WASHINGTON

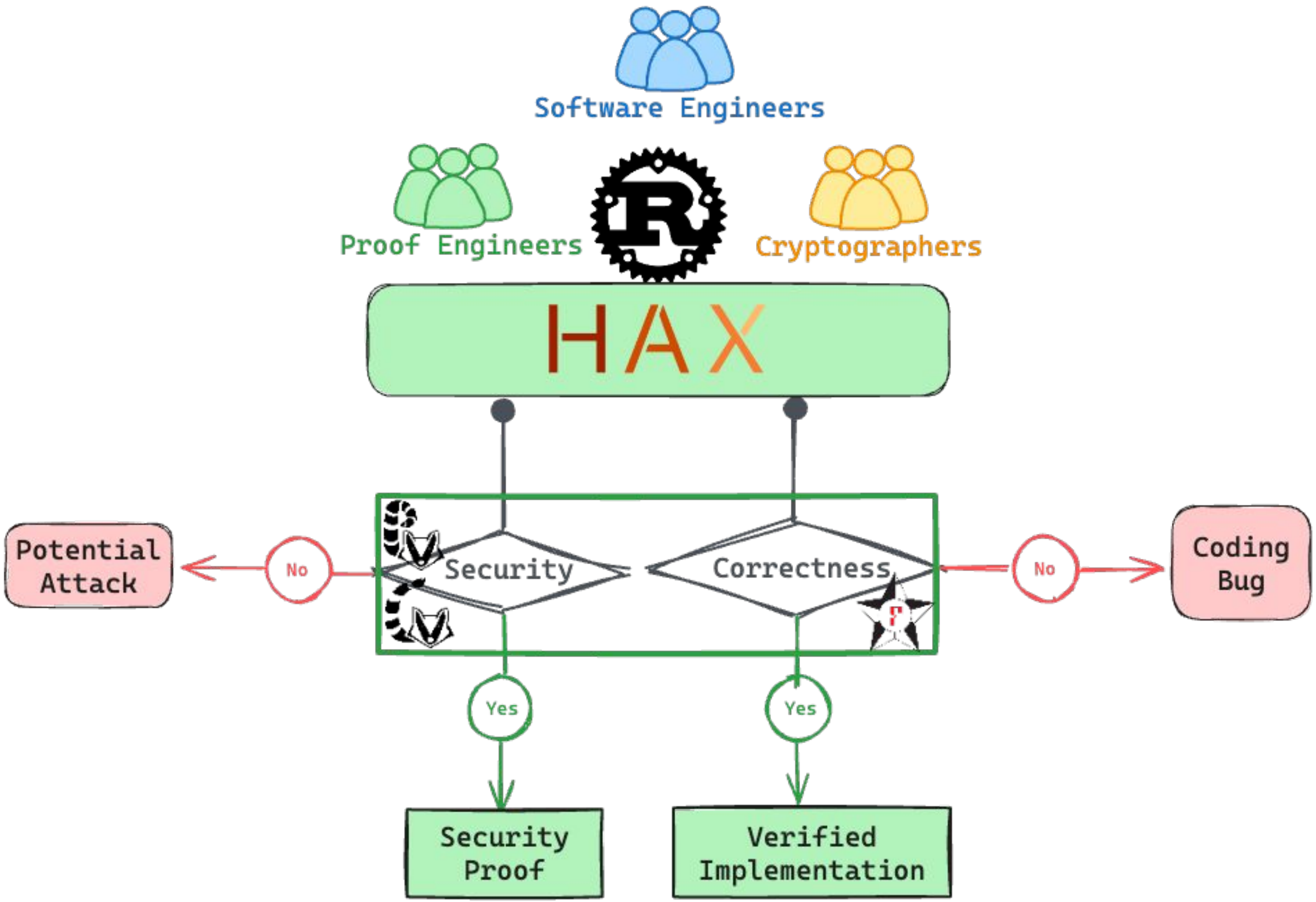
Usable Verification Tools

Formal Verification

Correctness

HAX

verification toolchain

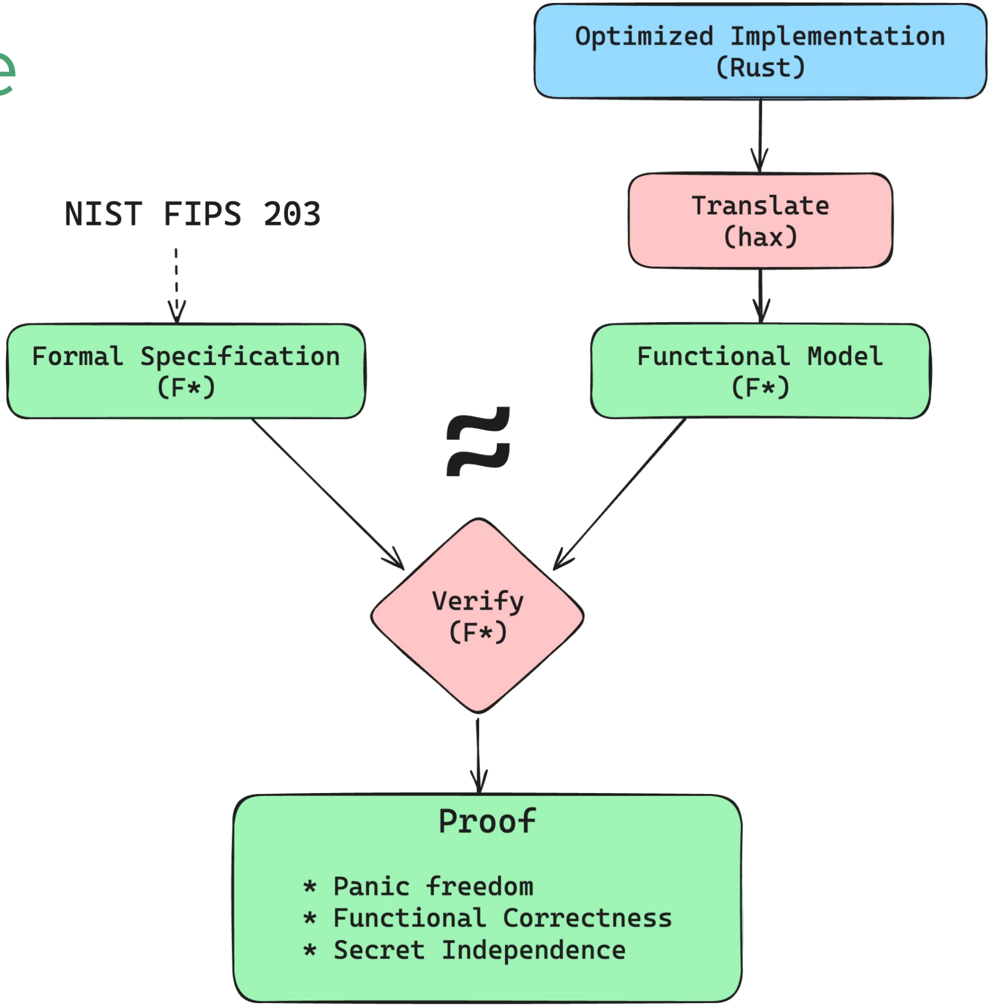


CRYSPEN

AARHUS UNIVERSITY

Inria

Verifying Rust Code with hax and F*



The hax process

hax: Process

```c
// reduce_once reduces 0 ⩽ x < 2*kPrime, mod kPrime.
static uint16_t reduce_once(uint16_t x) {
  assert(x < 2 * kPrime);
  const uint16_t subtracted = x - kPrime;
  uint16_t mask = 0u - (subtracted >> 15);
  // On Aarch64, omitting a |value_barrier_u16| results in a 2x speedup of Kyber
  // overall and Clang still produces constant-time code using `csel`. On other
  // platforms & compilers on godbolt that we care about, this code also
  // produces constant-time output.
  return (mask & x) | (~mask & subtracted);
}
```

```c
// constant time reduce x mod kPrime using Barrett reduction. x must be less
// than kPrime + 2×kPrime².
static uint16_t reduce(uint32_t x) {
  assert(x < kPrime + 2u * kPrime * kPrime);
  uint64_t product = (uint64_t)x * kBarrettMultiplier;
  uint32_t quotient = (uint32_t)(product >> kBarrettShift);
  uint32_t remainder = x - quotient * kPrime;
  return reduce_once(remainder);
}
```

hax: Process

```c
// reduce_once reduces 0 ⩽ x < 2*kPrime, mod kPrime.
static uint16_t reduce_once(uint16_t x) {
  assert(x < 2 * kPrime);
  const uint16_t subtracted = x - kPrime;
  uint16_t mask = 0u - (subtracted >> 15);
  // On Aarch64, omitting a |value_barrier_u16| results in a 2x speedup of Kyber
  // overall and Clang still produces constant-time code using `csel`. On other
  // platforms & compilers on godbolt that we care about, this code also
  // produces constant-time output.
  return (mask & x) | (~mask & subtracted);
}
```

```c
// constant time reduce x mod kPrime using Barrett reduction. x must be less
// than kPrime + 2×kPrime².
static uint16_t reduce(uint32_t x) {
  assert(x < kPrime + 2u * kPrime * kPrime);
  uint64_t product = (uint64_t)x * kBarrettMultiplier;
  uint32_t quotient = (uint32_t)(product >> kBarrettShift);
  uint32_t remainder = x - quotient * kPrime;
  return reduce_once(remainder);
}
```

hax: Process

```rust
#[requires(coefficient_bits <= 11 && i32::from(fe) <= FIELD_MODULUS)]
#[ensures(|result| result >= 0 && result <= (1 << coefficient_bits) - 1)]
pub(super) fn compress_q(coefficient_bits: usize, fe: u16) -> KyberFieldElement {
    let mut compressed: u32 = (fe as u32) << (coefficient_bits + 1);
    compressed += FIELD_MODULUS as u32;
    compressed /= (FIELD_MODULUS << 1) as u32;

    (compressed & ((1u32 << coefficient_bits) - 1)) as KyberFieldElement
}
```

1. Make the requirements formal
2. hax attributes for “design by contract”
3. F* statically checks that the properties hold

Example

Proving correctness

of Barrett reduction

Writing Crypto Code in Rust

```rust
pub(crate) fn barrett_reduce(input: i32) -> i32 {
    let t = (i64::from(input) * 20159) + (0x4_000_000 >> 1);
    let quotient = (t >> 26) as i32;
    let remainder = input - (quotient * 3329);
    remainder
}
```

Barrett Reduction: computes **input % 3329**
(in constant time)

Potential Panics in Rust Code

```
pub(crate) fn barrett_reduce(input: i32) -> i32 {
    let t = (i64::from(input) * 20159) + (0x4_000_000 >> 1);
    let quotient = (t >> 26) as i32;
    let remainder = input - (quotient * 3329);
    remainder
}
```

These arithmetic operations may overflow or underflow causing the code to panic at run-time

Proving Panic Freedom and Correctness in F*

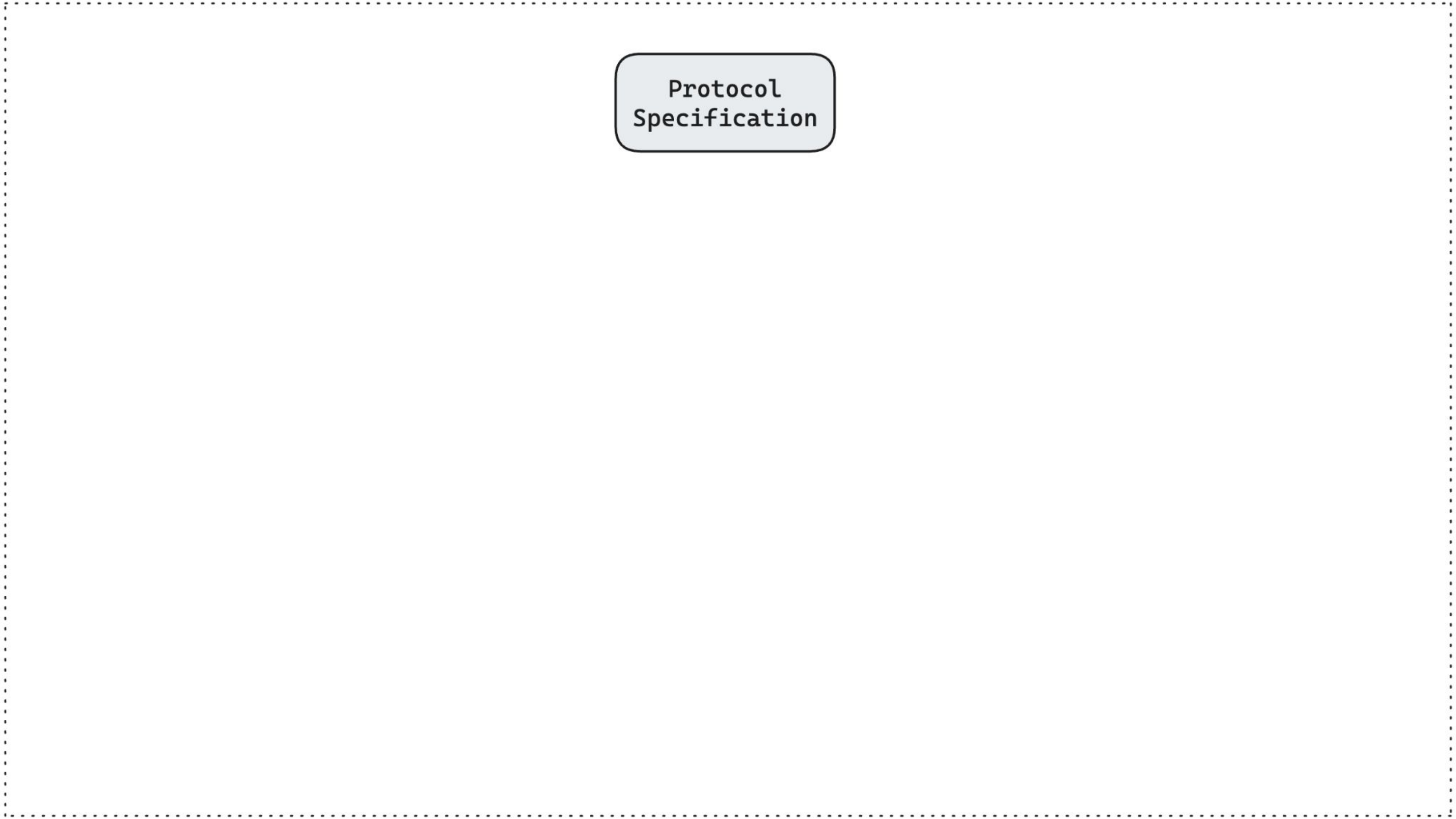
```
val barrett_reduce (input: i32_b (v v_BARRETT_R))
    : Pure (i32_b 3328)
    (requires True)
    (ensures fun result ->
        v result % v Libcrux.Kem.Kyber.Constants.v_FIELD_MODULUS
      = v  input %v Libcrux.Kem.Kyber.Constants.v_FIELD_MODULUS)
```

Expected behaviour: `result ≈ input % 3329`

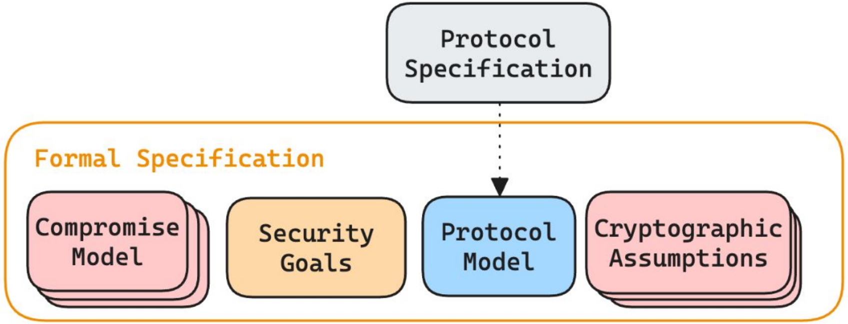
Formal Verification

Security

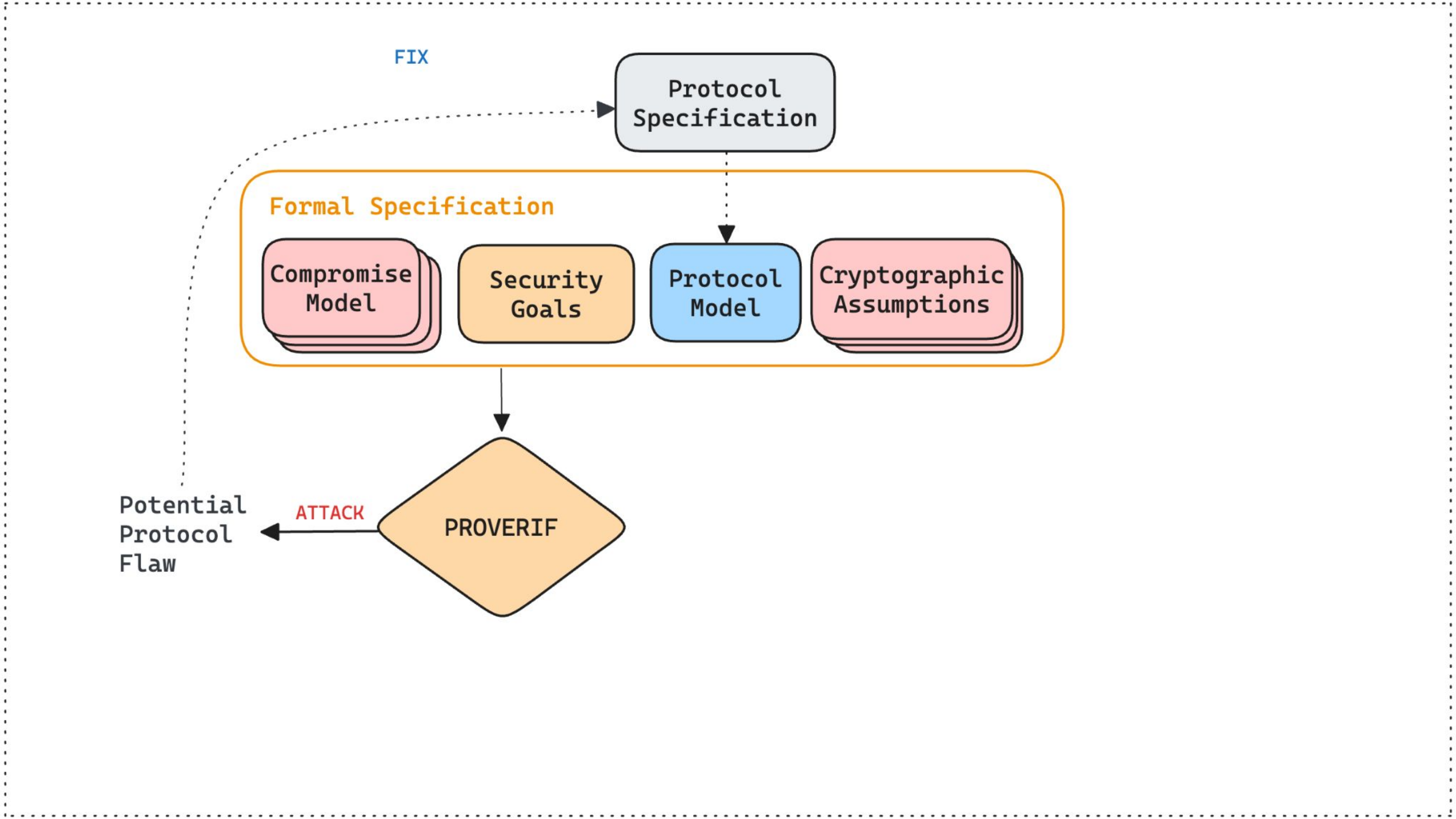
Our Formal Verification Methodology | Security



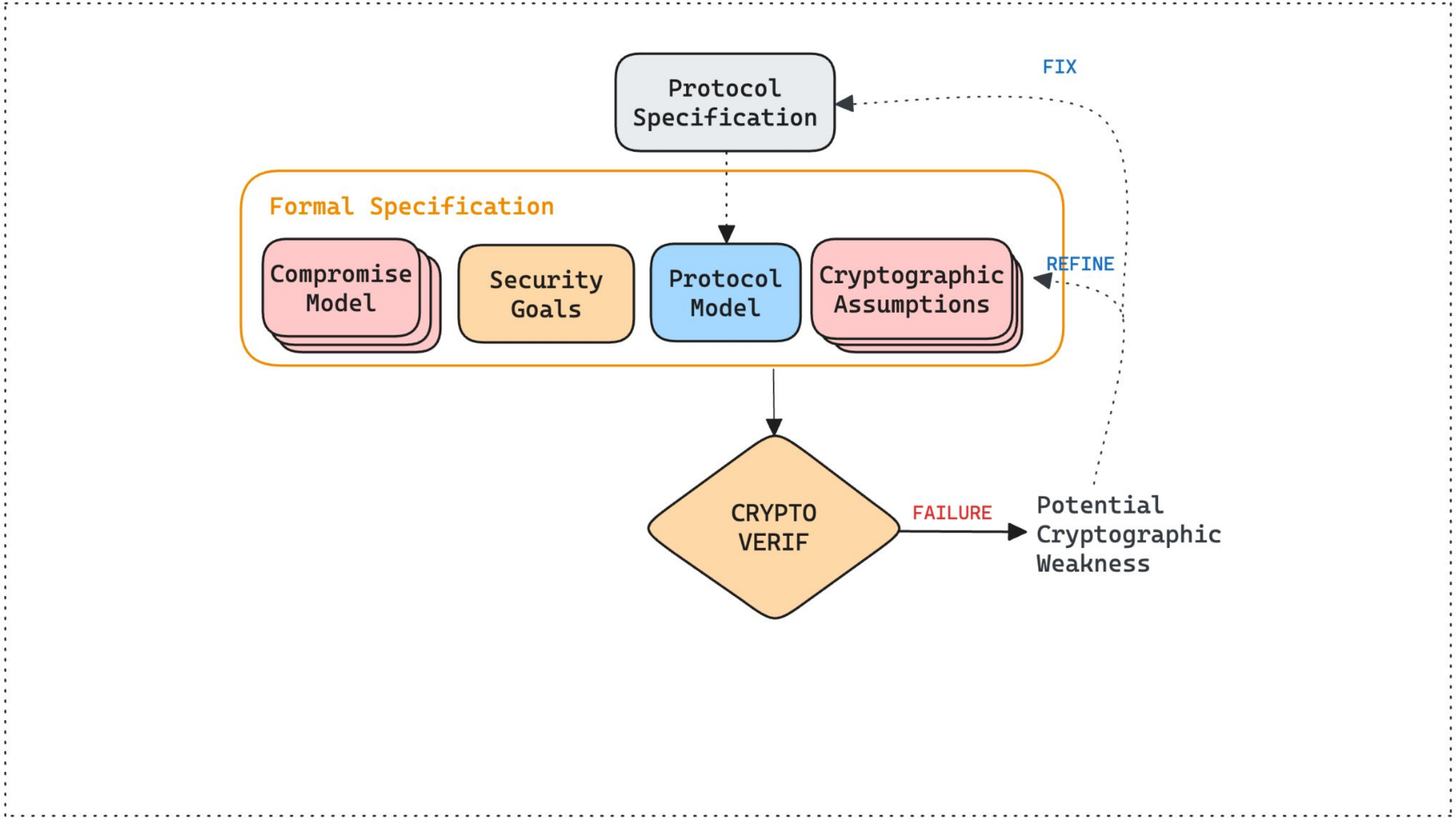
Our Formal Verification Methodology | Security



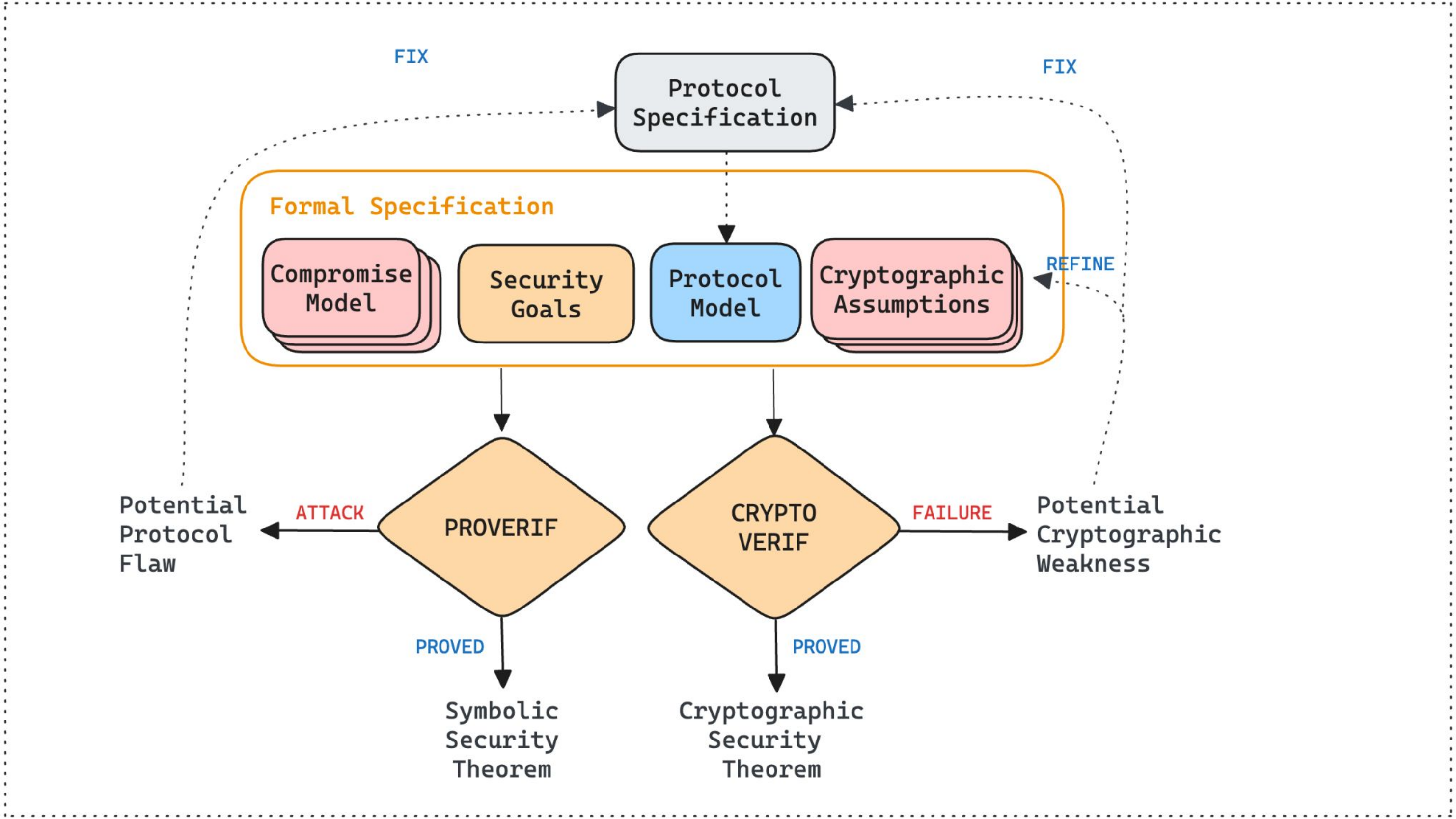
Our Formal Verification Methodology | Security



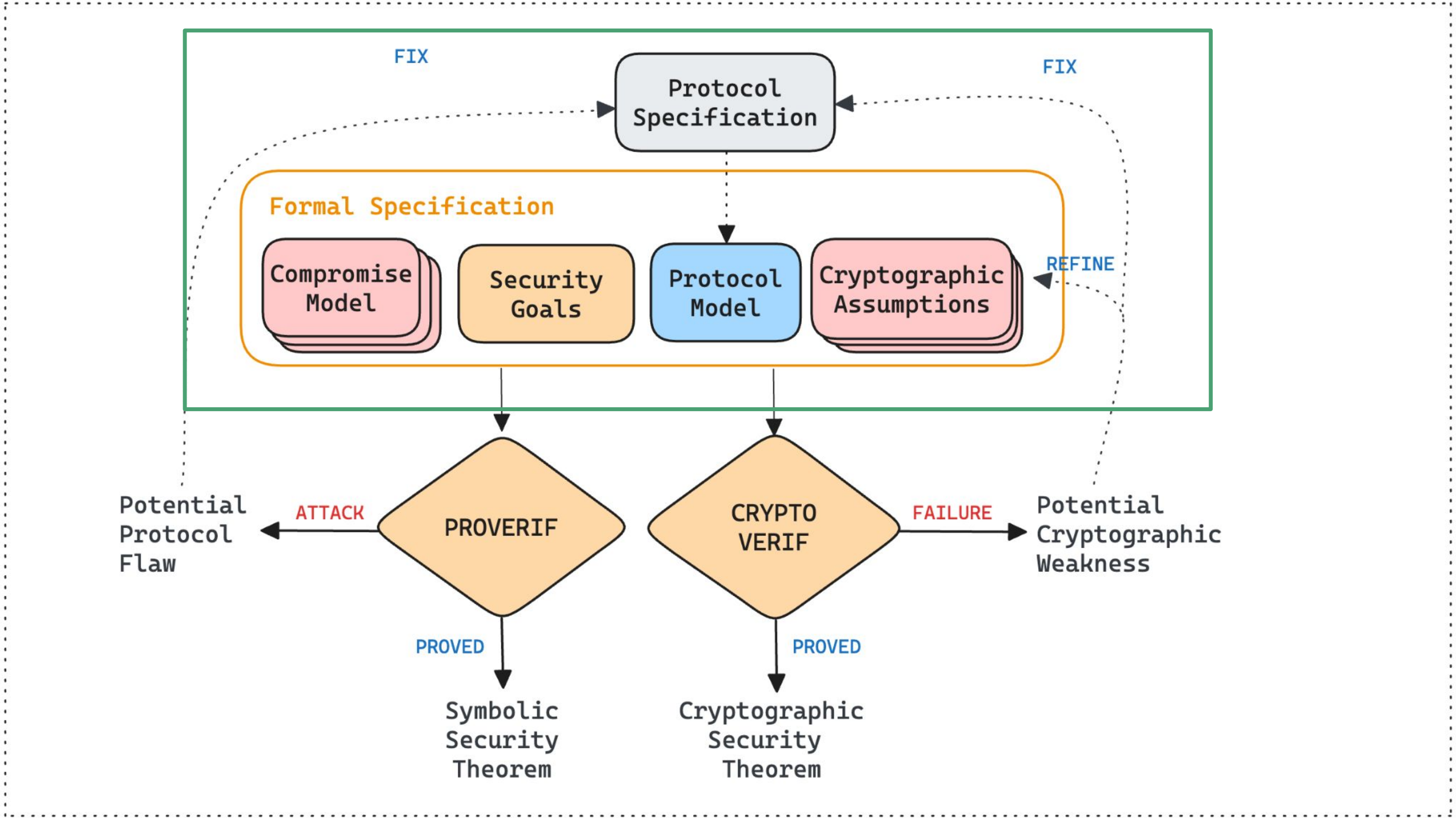
Our Formal Verification Methodology | Security



Our Formal Verification Methodology | Security



Our Formal Verification Methodology | Security



hax: ongoing projects

- **Rust Core:** an annotated version of the Rust Core library
- **Backends:** new backends for Lean, EasyCrypt, ProVerif

- **Verified**
  - **PQ Crypto:** verified Rust code for Kyber/ML-KEM, ...
  - **OS Modules:** verified kernel code for RIOT-OS
  - **Protocols:** verified code for EDHOC, MLS, TLS 1.3, ...
  - **Contracts:** verified canisters for Internet Computer

HAX

A Usable Tool for Verification

franziskus@cryspen.com