Formal Verification of Post-Quantum Cryptography in Formosa-Crypto

Manuel Barbosa mbb@fc.up.pt
University of Porto (FCUP) and INESC TEC and MPI-SP

Context and Goals

Computer-Aided Cryptography

- Take techniques from the study of programming languages such as:
  - Programming language design and compilation
  - Various approaches to program verification
  - Type systems for security
  - Interactive theorem provers
  - etc.

Different approaches tools technologies

SoK: Computer-Aided Cryptography

Manuel Barbosa*, Gilles Barthe†‡, Karthik Bhargavan§, Bruno Blanchet§, Cas Cremers¶, Kevin Liao†‖, Bryan Parno**
*University of Porto (FCUP) and INESC TEC, †Max Planck Institute for Security & Privacy, ‡IMDEA Software Institute, §INRIA Paris, ¶CISPA Helmholtz Center for Information Security, ‖MIT, **Carnegie Mellon University

***Abstract*—Computer-aided cryptography is an active area of research that develops and applies formal, machine-checkable approaches to the design, analysis, and implementation of cryptography. We present a cross-cutting systematization of the computer-aided cryptography literature, focusing on three main areas: (*i*) design-level security (both symbolic security and computational security), (*ii*) functional correctness and efficiency, and (*iii*) implementation-level security (with a focus on digital side-channel resistance). In each area, we first clarify the role of computer-aided cryptography—how it can help and what the caveats are—in addressing current challenges. We next present a taxonomy of state-of-the-art tools, comparing their accuracy, scope, trustworthiness, and usability. Then, we highlight their main achievements, trade-offs, and research challenges. After covering the three main areas, we present two case studies.**

which are difficult to catch by code testing or auditing; ad-hoc constant-time coding recipes for mitigating side-channel attacks are tricky to implement, and yet may not cover the whole gamut of leakage channels exposed in deployment. Unfortunately, the current modus operandi—relying on a select few cryptography experts armed with rudimentary tooling to vouch for security and correctness—simply cannot keep pace with the rate of innovation and development in the field.

Computer-aided cryptography, or *CAC* for short, is an active area of research that aims to address these challenges. It encompasses formal, machine-checkable approaches to designing, analyzing, and implementing cryptography; the variety of tools available address different parts of the problem space.

Computer-Aided Cryptography

- Apply them to (high-assurance) cryptography:
  - Domain-specific programming languages and compilers
  - Specification of crypto algorithms and protocols
  - Specification and analysis of security models
  - Formal verification of:
    - functional correctness
    - provable security
    - countermeasures against
      - side-channel attacks
      - micro-architectural attacks

Different
approaches
tools
technologies

SoK: Computer-Aided Cryptography

Manuel Barbosa*, Gilles Barthe†‡, Karthik Bhargavan§, Bruno Blanchet§, Cas Cremers¶, Kevin Liao†‖, Bryan Parno**
*University of Porto (FCUP) and INESC TEC, †Max Planck Institute for Security & Privacy, ‡IMDEA Software Institute,
§INRIA Paris, ¶CISPA Helmholtz Center for Information Security, ‖MIT, **Carnegie Mellon University

***Abstract*—Computer-aided cryptography is an active area of research that develops and applies formal, machine-checkable approaches to the design, analysis, and implementation of cryptography. We present a cross-cutting systematization of the computer-aided cryptography literature, focusing on three main areas: (*i*) design-level security (both symbolic security and computational security), (*ii*) functional correctness and efficiency, and (*iii*) implementation-level security (with a focus on digital side-channel resistance). In each area, we first clarify the role of computer-aided cryptography—how it can help and what the caveats are—in addressing current challenges. We next present a taxonomy of state-of-the-art tools, comparing their accuracy, scope, trustworthiness, and usability. Then, we highlight their main achievements, trade-offs, and research challenges. After covering the three main areas, we present two case studies.**

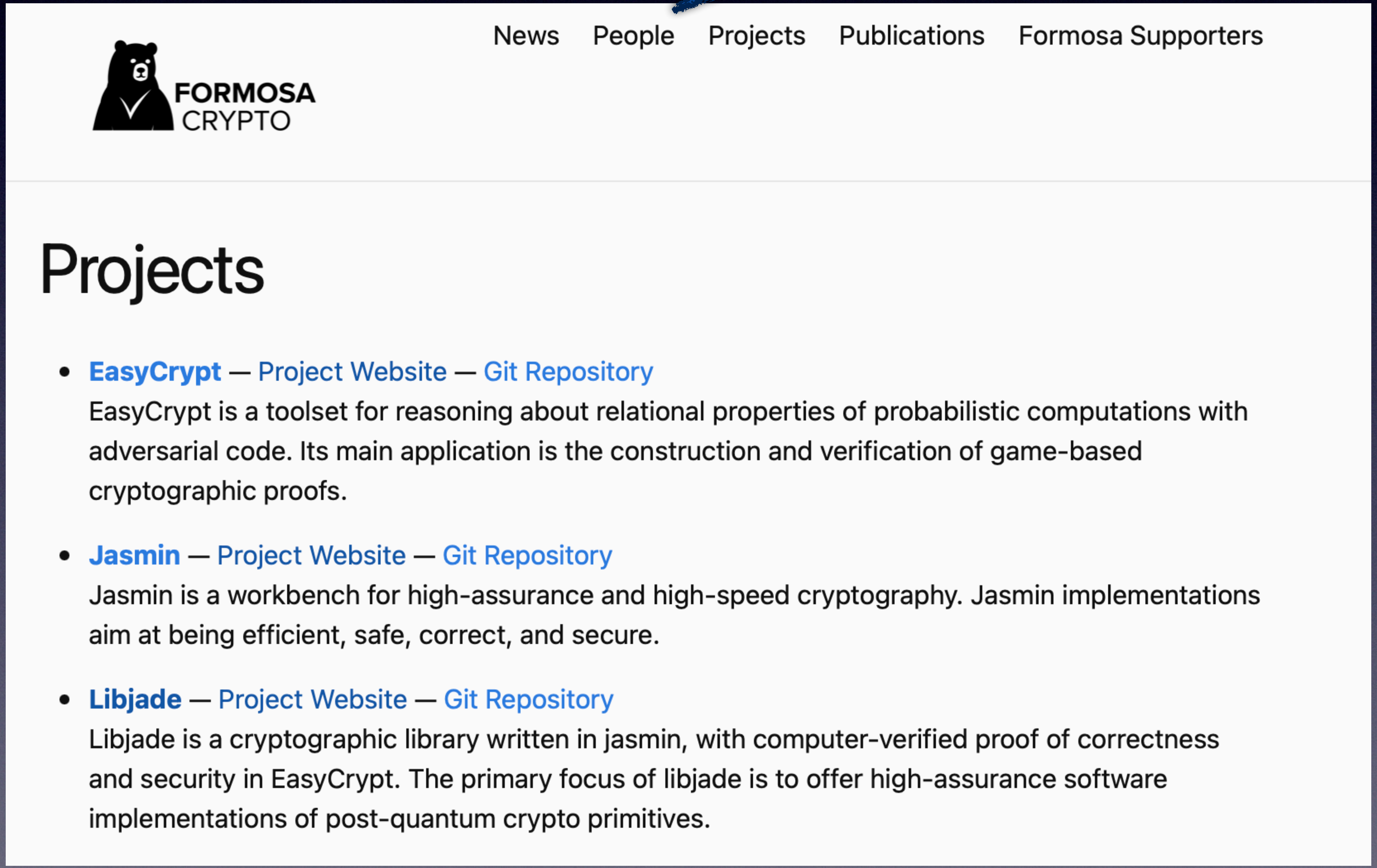
which are difficult to catch by code testing or auditing; ad-hoc constant-time coding recipes for mitigating side-channel attacks are tricky to implement, and yet may not cover the whole gamut of leakage channels exposed in deployment. Unfortunately, the current modus operandi—relying on a select few cryptography experts armed with rudimentary tooling to vouch for security and correctness—simply cannot keep pace with the rate of innovation and development in the field.

Computer-aided cryptography, or *CAC* for short, is an active area of research that aims to address these challenges. It encompasses formal, machine-checkable approaches to designing, analyzing, and implementing cryptography; the variety of tools available address different parts of the problem space.

Formosa Crypto

Community around Jasmin, EasyCrypt and libjade

- Access to tools, examples and usage guides
- Interact with developers and other users
- Learn what has been done and ongoing work
- Help understanding tools and solving problems
- Ask for new features
- Regular in person meetings:
  - Jasmin/EasyCrypt/libjade development
  - research projects around the tools
  - investigate new ideas, collaborations


News People Projects Publications Formosa Supporters

FORMOSA CRYPTO

Projects

- **EasyCrypt** — Project Website — Git Repository
  EasyCrypt is a toolset for reasoning about relational properties of probabilistic computations with adversarial code. Its main application is the construction and verification of game-based cryptographic proofs.
- **Jasmin** — Project Website — Git Repository
  Jasmin is a workbench for high-assurance and high-speed cryptography. Jasmin implementations aim at being efficient, safe, correct, and secure.
- **Libjade** — Project Website — Git Repository
  Libjade is a cryptographic library written in jasmin, with computer-verified proof of correctness and security in EasyCrypt. The primary focus of libjade is to offer high-assurance software implementations of post-quantum crypto primitives.

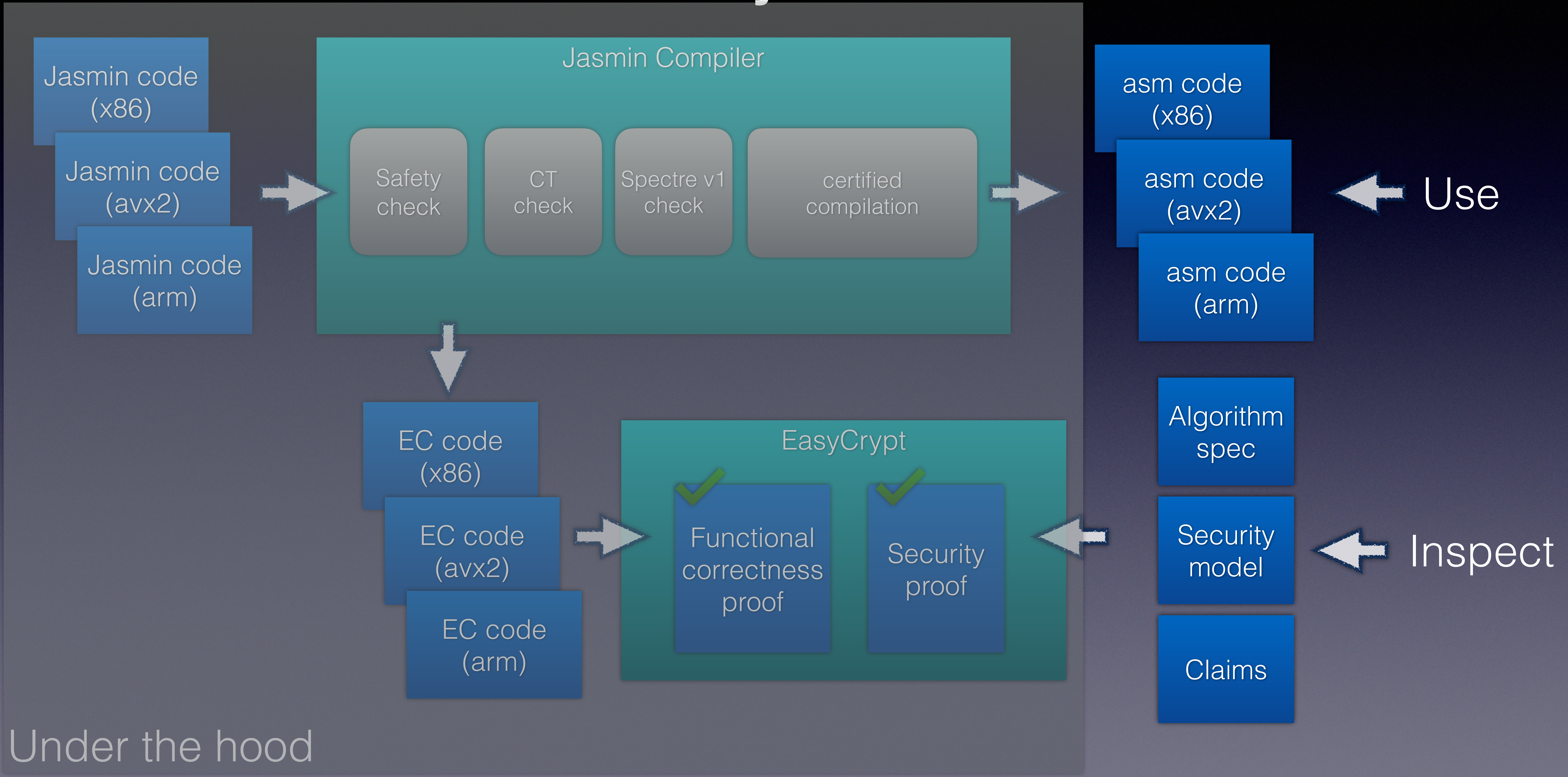
Interactively in a Zulip server

formosa-crypto.org

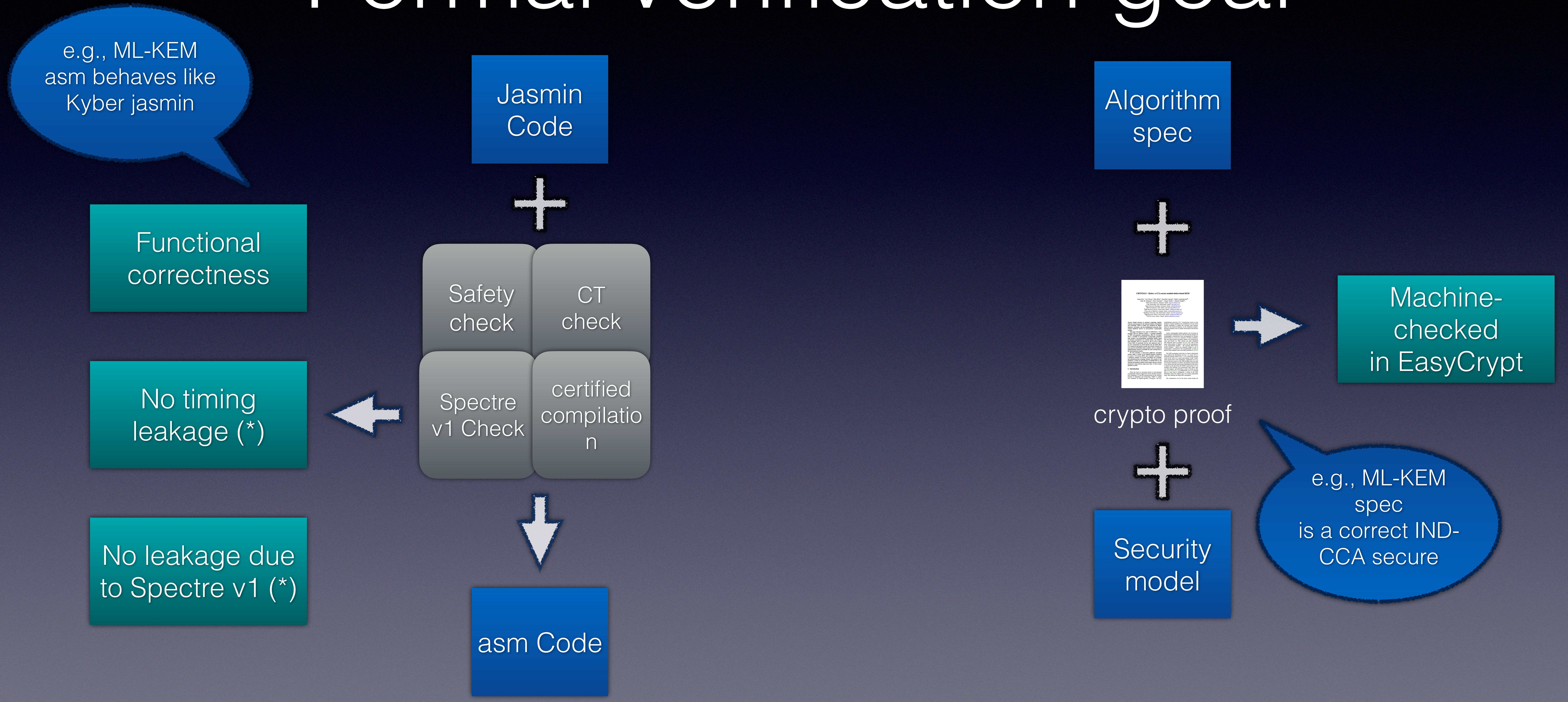
libjade

- Open-source high-assurance cryptographic library (SUPERCOP-like C API)
- Current features:
  - High-speed implementations for AMD64 (aka x86_64 or x64 + AVX2) and ARMv7 (32-bit)
  - Cryptographic hash functions and XOFs (SHA-2, SHA-3, SHAKE)
  - One-time authenticators and stream ciphers (poly1305, ChaCha, Salsa)
  - Authenticated encryption (XSalsa20Poly1305)
  - Curve 25519
  - Postquantum KEM and Signature (ML-KEM, ML-DSA, SLH-DSA)

libjade

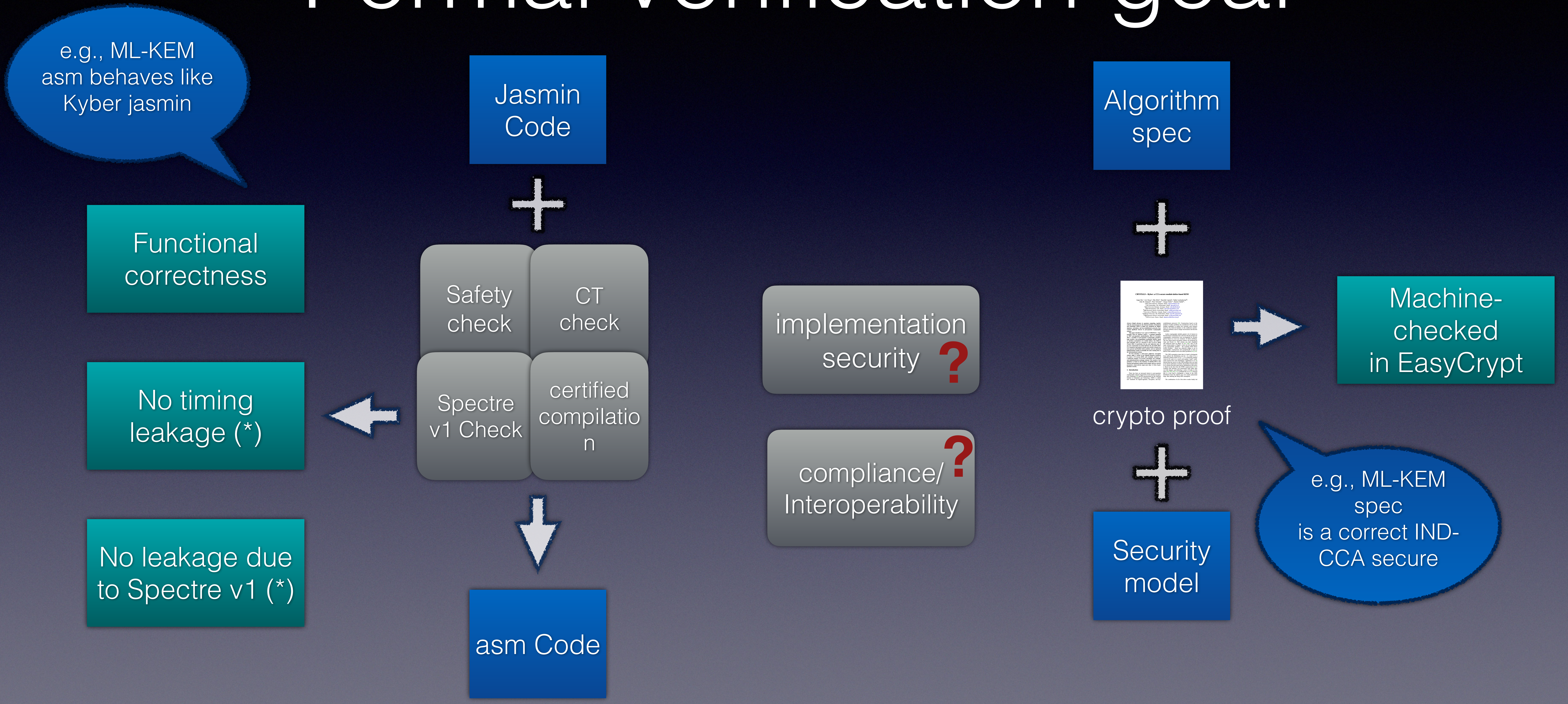

Jasmin code (x86)
Jasmin code (avx2)
Jasmin code (arm)
Jasmin Compiler
Safety check
CT check
Spectre v1 check
certified compilation
asm code (x86)
asm code (avx2)
asm code (arm)
Use
EC code (x86)
EC code (avx2)
EC code (arm)
EasyCrypt
Functional correctness proof
Security proof
Algorithm spec
Security model
Claims
Inspect
Under the hood

Formal verification goal



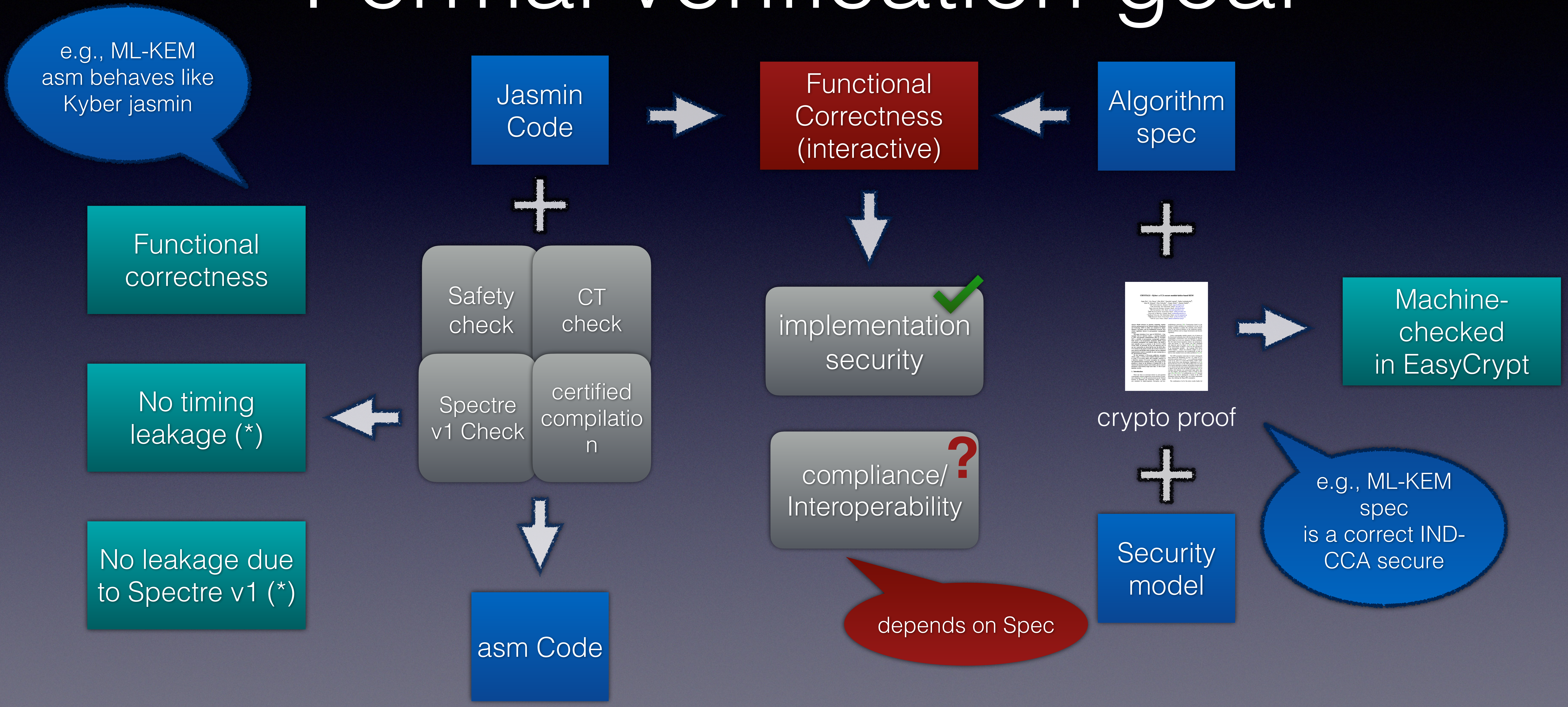
(*) in a formally defined (abstract) leakage model

Formal verification goal



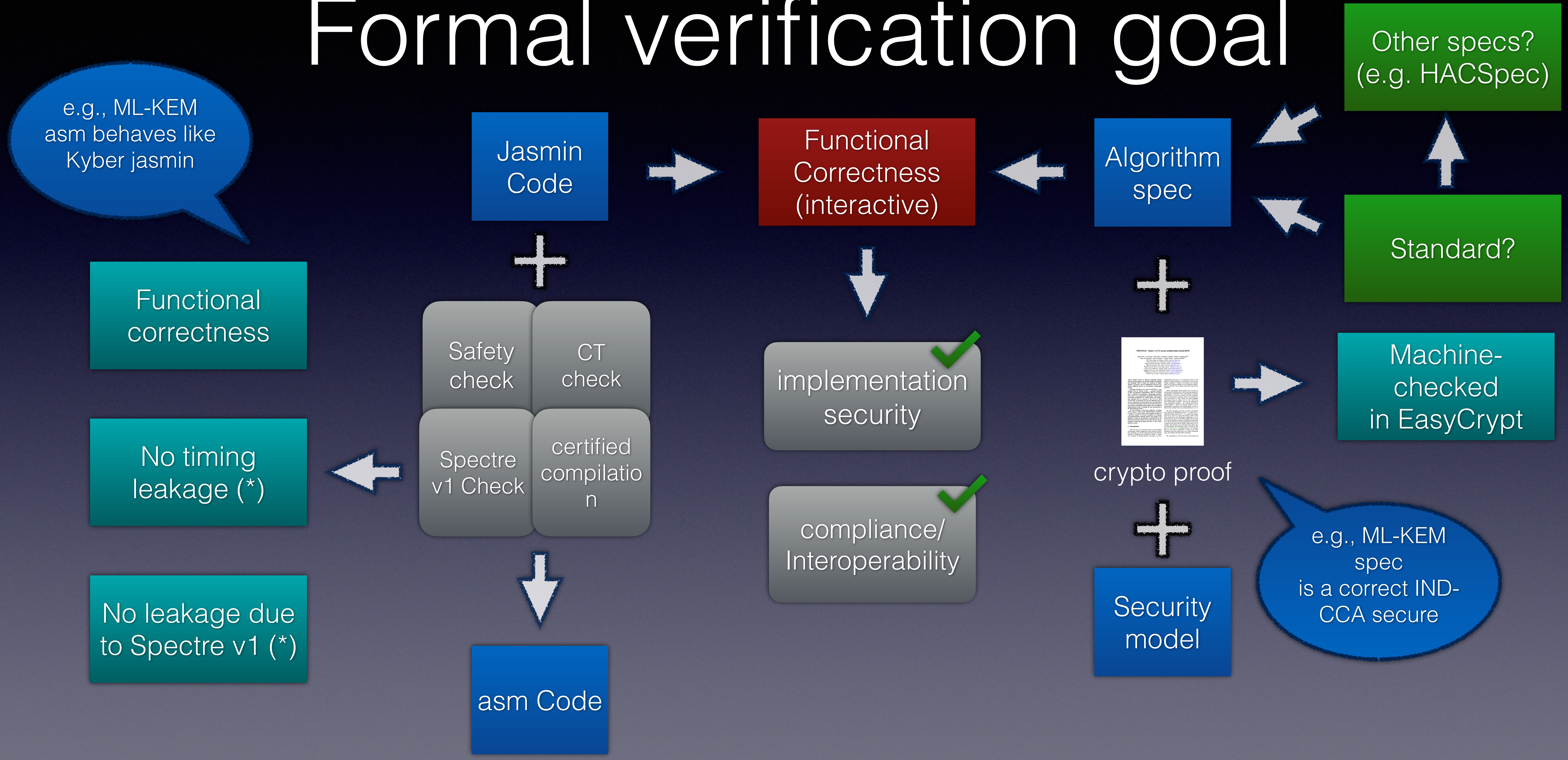
(*) in a formally defined (abstract) leakage model

Formal verification goal



(*) in a formally defined (abstract) leakage model

Formal verification goal



(*) in a formally defined (abstract) leakage model

Jasmin Programming

Jasmin: Goals

- Empower programmers to deliver fast and formally verified assembly code
  - Efficiency & verification-friendly source language
  - Efficiency & provably property -checking/-preserving compiler (safety, functional correctness, protection against timing attacks)
  - Verification infrastructure (based on EasyCrypt):
    - functional correctness wrt high-level spec
    - provable security wrt to formal (computational) cryptographic model

Jasmin: Zero cost abstractions

```
inline fn init(reg u64 key nonce, reg u32 counter) → stack u32[16]
{
  inline int i;
  stack u32[16] st;
  reg u32[8] k;
  reg u32[3] n;

  st[0] = 0x61707865;
  st[1] = 0x3320646e;
  st[2] = 0x79622d32;
  st[3] = 0x6b206574;

  for i=0 to 8 {
    k[i] = (u32)[key + 4*i];
    st[4+i] = k[i];
  }

  st[12] = counter;

  for i=0 to 3 {
    n[i] = (u32)[nonce + 4*i];
    st[13+i] = n[i];
  }

  return st;
}
```

- Things one wishes asm could offer:
  - Variable names instead of registers
  - Arrays: collections of variables
  - Automatic stack management
  - Readable loop structures
  - (inlineable) function calls
  - nice syntax and clever type checking

Jasmin: Zero cost abstractions

```
inline fn init(reg u64 key nonce, reg u32 counter) → stack u32[16]
{
  inline int i;
  stack u32[16] st;
  reg u32[8] k;
  re

  st
  st
  st
  st

  fo
    k[i] = (u32)[key + 4*i];
    st[4+i] = k[i];
  }

  st[12] = counter;

  for i=0 to 3 {
    n[i] = (u32)[nonce + 4*i];
    st[13+i] = n[i];
  }

  return st;
}
```

- Things one wishes asm could offer:
  - Variable names instead of registers
  - es
  - s
  - nice syntax and clever type checking

Programmer knows what assembly is going to look like: one-to-one instruction translation

We call this "asm in the head"
(qhasm inspiration)

Jasmin: per arch instruction set

```
inline
fn __csubq(reg u256 r qx16) -> reg u256
{
  reg u256 t;
  r = #VPSUB_16u16(r, qx16);
  t = #VPSRA_16u16(r, 15);
  t = #VPAND_256(t, qx16);
  r = #VPADD_16u16(t, r);
  return r;
}
```

```
fn _poly_csubq(reg ptr u16[KYBER_N] rp) -> reg ptr u16[KYBER_N]
{
  reg u64 i;
  reg u16 t;
  reg u16 b;

  i = 0;
  while (i < KYBER_N)
  {
    t = rp[(int)i];
    t -= KYBER_Q;
    b = t;
    b >>s= 15;
    b &= KYBER_Q;
    t += b;
    rp[(int)i] = t;
    i += 1;
  }
  return rp;
}
```

- Common instructions
  - nice syntax (same across architectures)
- All instructions
  - available via instruction name
- Support for all word sizes
- No memory allocation
  - caller allocates memory

Jasmin: per arch instruction set

```
inline
fn __csubq(reg u256 r qx16) -> reg u256
{
  reg u256 t;
  r = #VPSUB_16u16(r, qx16);
  t = #VPSRA_16u16(r, 15);
  t = #VPAND_256(t, qx16);
  r = #VPADD_16
  return r;
}
```

```
fn _poly_csubq(reg ptr u16[KYBER_N] rp) -> reg ptr u16[KYBER_N]
{
  reg u64 i;
  reg u16 t;
  reg u16 b;

  i = 0;
  while (i < KYBER_N)
  {
    t = rp[(int)i];
    t -= KYBER_Q;
    b = t;
    b >>s= 15;
    b &= KYBER_Q;
    t += b;
    rp[(int)i] = t;
    i += 1;
  }
  return rp;
}
```

- Common instructions
  - nice syntax (same across architectures)
  - available via instruction name
- No memory allocation
  - caller allocates memory

Programmer responsible for all spilling

Compilation breaks if register assignment not found.

Jasmin: per arch instruction set

```
inline
fn __csubq(reg u256 r qx16) -> reg u256
{
  reg u256 t;
  r = #VPSUB_16u16(r, qx16);
  t = #VPSRA_16u16(r, 15);
  t = #VPAND_256(t, qx16);
  r = #VPADD_16u16(t, r);
  return r;
}
```

```
fn _poly_csubq(reg ptr u16[KYBER_N] rp) -> reg ptr u16[KYBER_N]
{
  reg u64 i;
  reg u16 t;
  reg u16 b;

  i = 0;
  while (i < KYBER_N)
  {
    t = rp[(int)i];
    t -= KYBER_Q;
    b = t;
    b >>s= 15;
    b &= KYBER_Q;
    t += b;
    rp[(int)i] = t;
    i += 1;
  }
  return rp;
}
```

- Internal function calls:
  - arbitrary calling convention
  - global reg allocation
  - restricted pointers: stack regions
- External entry points
  - standard ABI/calling convention

Jasmin: per arch instruction set

```
inline
fn __csubq(reg u256 r qx16) -> reg u256
{
  reg u256 t;
  r = #VPSUB_16u16(r, qx16);
  t = #VPSRA_16u16(r, 15);
  t = #VPAND_256(t, qx16);
  r = #VPADD_1
  return r;
}
```

```
fn _poly_csubq(reg ptr u16[KYBER_N] rp) -> reg ptr u16[KYBER_N]
{
  reg u64 i;
  reg u16 t;
  reg u16 b;

  i = 0;
  while (i < KYBER_N)
  {
    t = rp[(int)i];
    t -= KYBER_Q;
    b = t;
    b >>s= 15;
    b &= KYBER_Q;
    t += b;
    rp[(int)i] = t;
    i += 1;
  }
  return rp;
}
```

- Internal function calls:
  - arbitrary calling convention
  - global reg allocation
  - restricted pointers: stack regions
- External entry points
  - standard ABI/calling convention

Good documentation and error msgs …

… are work in progress.

Jasmin: per arch instruction set

```
inline
fn __csubq(reg u256 r qx16) -> reg u256
{
  reg u256 t;
  r = #VPSUB_16u16(r, qx16);
  t = #VPSRA_16u16(r, 15);
  t = #VPAND_256(t, qx16);
  r = #VPADD_16u16(t, r);
  return r;
}
```

```
fn _poly_csubq(reg ptr u16[KYBER_N] rp) -> reg ptr u16[KYBER_N]
{
  reg u64 i;
  reg u16 t;
  reg u16 b;

  i = 0;
  while (
  {
    t = r
    t -= KYBER_Q;
    b = t;
    b >>s= 15;
    b &= KYBER_Q;
    t += b;
    rp[(int)i] = t;
    i += 1;
  }
  return rp;
}
```

- Internal function calls:
  - arbitrary calling convention
  - global reg allocation
  - restricted pointers: stack regions
- External entry points
  - standard ABI/calling convention

Zulip server is a good friend!

Q&A log really helps other users/developers.

EasyCrypt Verification

EasyCrypt

- Two languages: functional (define operators), imperative (implement algorithms)
- Logics to reason about properties of
  - real values (probabilities), distributions, etc.
  - functional programs (operators)
  - imperative programs (probabilistic Hoare logic or pHL)
  - relations between two imperative programs (probabilistic pHL or pRHL)
- These logics are interconnected:
  - use logic A to discharge side-conditions of logic B proof steps
  - prove claims in logic A using (a combination of) other logic(s)

(Prob) Hoare logic

```
module M = {
  var v1 : int
  var v2 : int

  proc f(x:int; y: int) = {
    v1 ← 0;
    return x + y;
  }

  proc g(x:int) = {
    v1 ← 0;
    return 2*x;
  }
}.
```

- Classical Hoare triple based on two predicates
  - Precondition: assumed in starting state
  - Postcondition: ensured in final state

```
lemma relate : ∀ _x _y _v2, hoare[M.f : arg=(_x,_y) ∧ M.v2 = _v2 ⟹ res=_x + _y ∧ M.v2=_v2].
```

(Prob) Hoare logic

```
module M = {
  var v1 : int
  var v2 : int

  proc f(x:int; y: int) = {
    v1 ← 0;
    return x + y;
  }

  proc g(x:in
    v1 ← 0;
    return 2*x;
  }
}.
```

- Your usual Hoare triple based on two predicates

Initially: prove that some event is rare

- Postcondition: ensured in final state

lemma relate : $\forall$ _x _y _v2, hoare[M.f : arg=(_x,_y) $\wedge$ M.v2 = _v2 $\Longrightarrow$ res=_x + _y $\wedge$ M.v2=_v2].

(Prob) Hoare logic

```
module M = {
  var v1 : int
  var v2 : int

  proc f(x:int; y: int) = {
    v1 ← 0;
    return x + y;
  }

  proc g(x:int) = {
    v1 ← 0;
    return 2∗x;
  }
}.
```

predicates

state

te

Very useful: prove that procedures implement convenient functional specs

```
lemma relate : ∀ _x _y _v2, hoare[M.f : arg=(_x,_y) ∧ M.v2 = _v2 ⟹ res=_x + _y ∧ M.v2=_v2].
```

(Prob) Hoare logic

```
module M = {
  var v1 : int
  var v2 : int

  proc f(x:int; y: int) = {
    v1 ← 0;
    return x + y;
  }

  proc g(x:int) = {
    v1 ← 0;
    return 2*x;
  }
}.
```

o predicates

state

te

Very useful: prove that procedures implement convenient functional specs

e.g., Jasmin code implements inner product correctly

lemma relate : ∀ _x _y _v2, hoare[M.f : arg=(_x,_y) ∧ M.v2 = _v2 ⟹ res=_x + _y ∧ M.v2=_v2].

(Prob) Relational Hoare logic

```
module M = {
  var v1 : int
  var v2 : int

  proc f(x:int; y: int) = {
    v1 ← 0;
    return x + y;
  }

  proc g(x:int) = {
    v1 ← 0;
    return 2∗x;
  }
}.
```

- Property that relates the behavior of two programs
  - Precondition: relation between starting states
  - Postcondition: relation between final states

```
equiv relate _x : M.f ∼ M.g : arg{1}=(_x,_x) ∧ arg{2} = _x ⟹ ={res}.
```

(Prob) Relational Hoare logic

```
module M = {
  var v1 : int
  var v2 : int

  proc f(x:int; y: int) =
    v1 ← 0;
    return x + y;
  }

  proc g(x:int) = {
    v1 ← 0;
    return 2*x;
  }
}.
```

In general: used to prove that two programs are equivalent, possibly up to bad.

programs

g states

tates

```
equiv relate _x : M.f ∼ M.g : arg{1}=(_x,_x) ∧ arg{2} = _x ⟹ ={res}.
```

(Prob) Relational Hoare logic

```
module M = {
  var v1 : int
  var v2 : int

  proc f(x:i
    v1 ← 0;
    return x
  }

  proc g(x:i
    v1 ← 0;
    return 2*x;
  }
}.
```

- Property that relates the behavior of two programs

Very useful: prove that two implementations are equivalent.

es

  - Postcondition: relation between final states

spec vs implementation

```
equiv relate _x : M.f ~ M.g : arg{1}=(_x,_x) ∧ arg{2} = _x ⟹ ={res}.
```

(Prob) Relational Hoare logic

```
module M = {
  var v1 : int
  var v2 : int

  proc f(x:i
    v1 ← 0;
    return x
  }

  proc g(x:i
    v1 ← 0;
    return 2*x;
  }
}.
```

- Property that relates the behavior of two programs

es

  - Postcondition: relation between final states

Very useful: prove that two implementations are equivalent.

implementation vs optimized implementation

```
equiv relate ... res}.
```

How does a proof in EC look like?

- Program/script
  - Convince tool that claim holds
  - Guiding it step by step to this conclusion
  - Using a set of rules/results that it knows are correct
  - Often relying on smt solver which EasyCrypt trusts

```
lemma add_corr (a b : W16.t) (a' b' : Fq) (asz bsz : int):
   0 <= asz < 15 => 0 <= bsz < 15 =>
   a' = inFq (W16.to_sint a) =>
   b' = inFq (W16.to_sint b) =>
   bw16 a asz =>
   bw16 b bsz =>
     inFq (W16.to_sint (a + b)) = a' + b' /\
           bw16 (a + b) (max asz bsz + 1).
proof.
pose aszb := 2^asz.
pose bszb := 2^bsz.
move => /= *.
have /= bounds_asz : 0 < aszb <= 2^14
 by split; [ apply gt0_pow2
             | move => *; rewrite  /aszb; apply StdOrder.IntOrder.ler_weexpn2l => /> /#].
have /= bounds_bsz : 0 < bszb <= 2^14
 by split; [ apply gt0_pow2
             | move => *; rewrite  /bszb; apply StdOrder.IntOrder.ler_weexpn2l => /> /#].
rewrite !to_sintD_small => />; first  by smt().
split; 1: by smt(inFqD).
rewrite (Ring.IntID.exprS 2 (max asz bsz)); 1: by smt().
by smt(exp_max).
qed.
```

Where we are

SHA3 (former Keccak)

- Security proof ✅
  - Indifferentiability from RO (classical)
  - Generic results for Sponge
- Implementation
  - AMD64 ✅
  - AVX2 ✅
  - ARMv7 ✅
- Functional correctness
  - AMD64 ✅
  - AVX2 ✅
  - ARMv7 🚧

ML-KEM (former Kyber)

- Security proof ✅
  - IND-CCA in the ROM (classical)
  - Generic results for Fujisaki-Okamoto transform
- Implementation
  - AMD64 ✅
  - AVX2 ✅
  - ARMv7 ✅
- Functional correctness
  - AMD64 ✅
  - AVX2 ✅
  - ARMv7 🚧

ML-DSA (former Dilithium)

- Security proof ✅
  - UF-CMA in ROM (classical)
  - Generic results for FS with aborts
- Implementation
  - AMD64 ✅
  - AVX2 ✅
  - ARMv7 ✅
- Functional correctness
  - AMD64 🚧
  - AVX2 🚧
  - ARMv7 🚧

SLH-DSA (former SPHINCS+)

- Security proof ✅
  - UF-CMA (classical)
  - Generic results for Hash-based signatures

- Implementation
  - AMD64 ✅
  - AVX2
  - ARMv7

- Functional correctness
  - AMD64 🚧
  - AVX2
  - ARMv7

X-Wing (Hybrid KEM)

- Security proof ✅
  - IND-CCA in the ROM (classical)
  - Builds on ML-KEM, x25519 and SHA3

- Implementation
  - AMD64
  - AVX2 ✅
  - ARMv7

- Functional correctness
  - AMD64
  - AVX2 🚧
  - ARMv7

The End

Questions?