Formally Verifying Kyber
Part I: Functional Correctness

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

Joint work with

- José Bacelar Almeida
- Gilles Barthe
- Benjamin Grégoire
- Vincent Laporte
- Jean-Christophe Léchenet
- Tiago Oliveira
- Hugo Pacheco
- Miguel Quaresma
- Peter Schwabe
- Antoine Séré
- Pierre-Yves Strub

https://eprint.iacr.org/2023/215

Accepted for publication at TCHES 2023

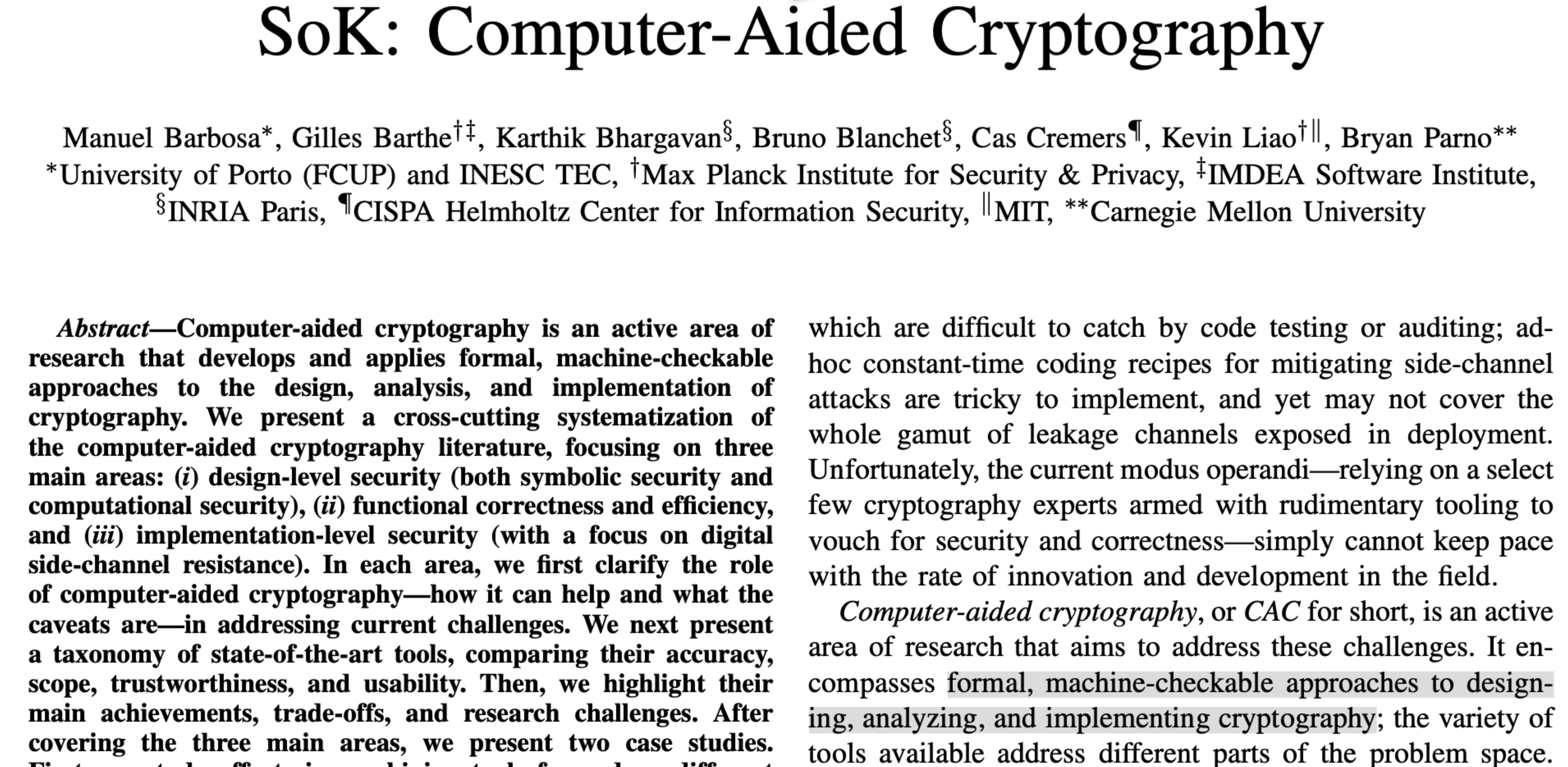
The Big Picture

- Computer Aided Cryptography
- Formosa Crypto initiative
- libjade project

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.**

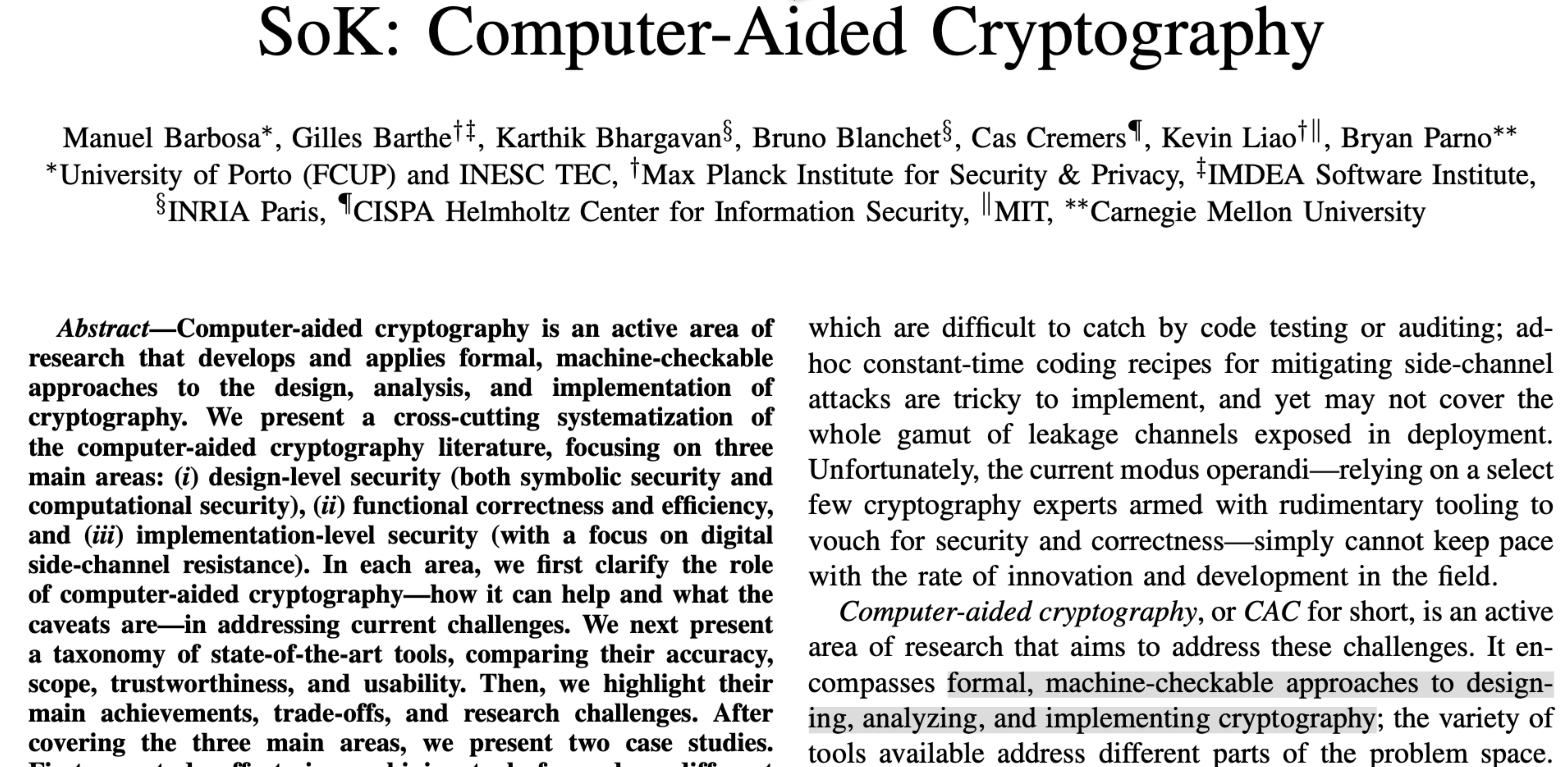
which are difficult to catch by code testing or auditing; ad-hoc constant-time coding recipes for mitigating side-channel attacks are tricky to implement, and yet may not cover the whole gamut of leakage channels exposed in deployment. Unfortunately, the current modus operandi—relying on a select few 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.**

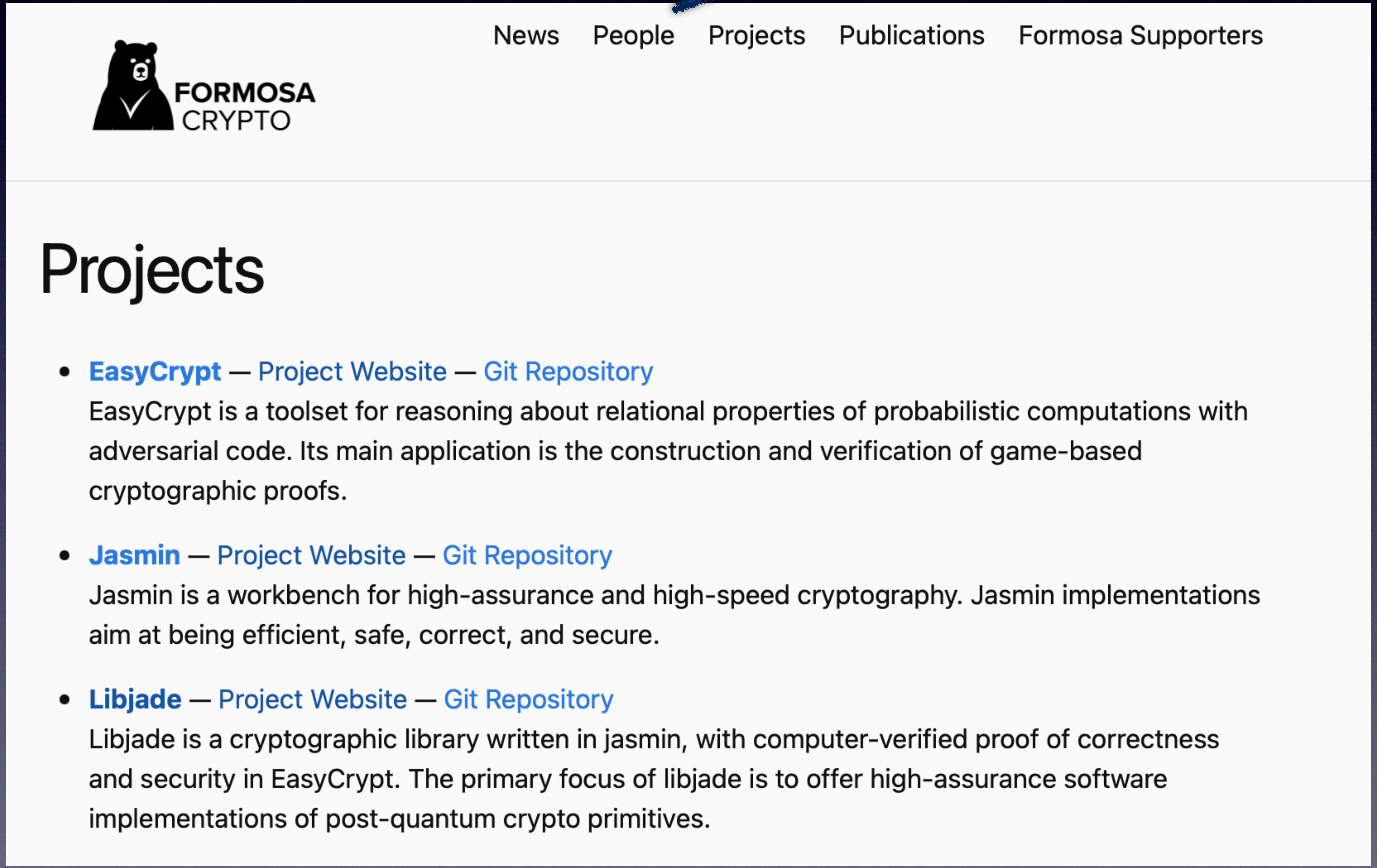
which are difficult to catch by code testing or auditing; ad-hoc constant-time coding recipes for mitigating side-channel attacks are tricky to implement, and yet may not cover the whole gamut of leakage channels exposed in deployment. Unfortunately, the current modus operandi—relying on a select few 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

- 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

Community around Jasmin, EasyCrypt and libjade



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.

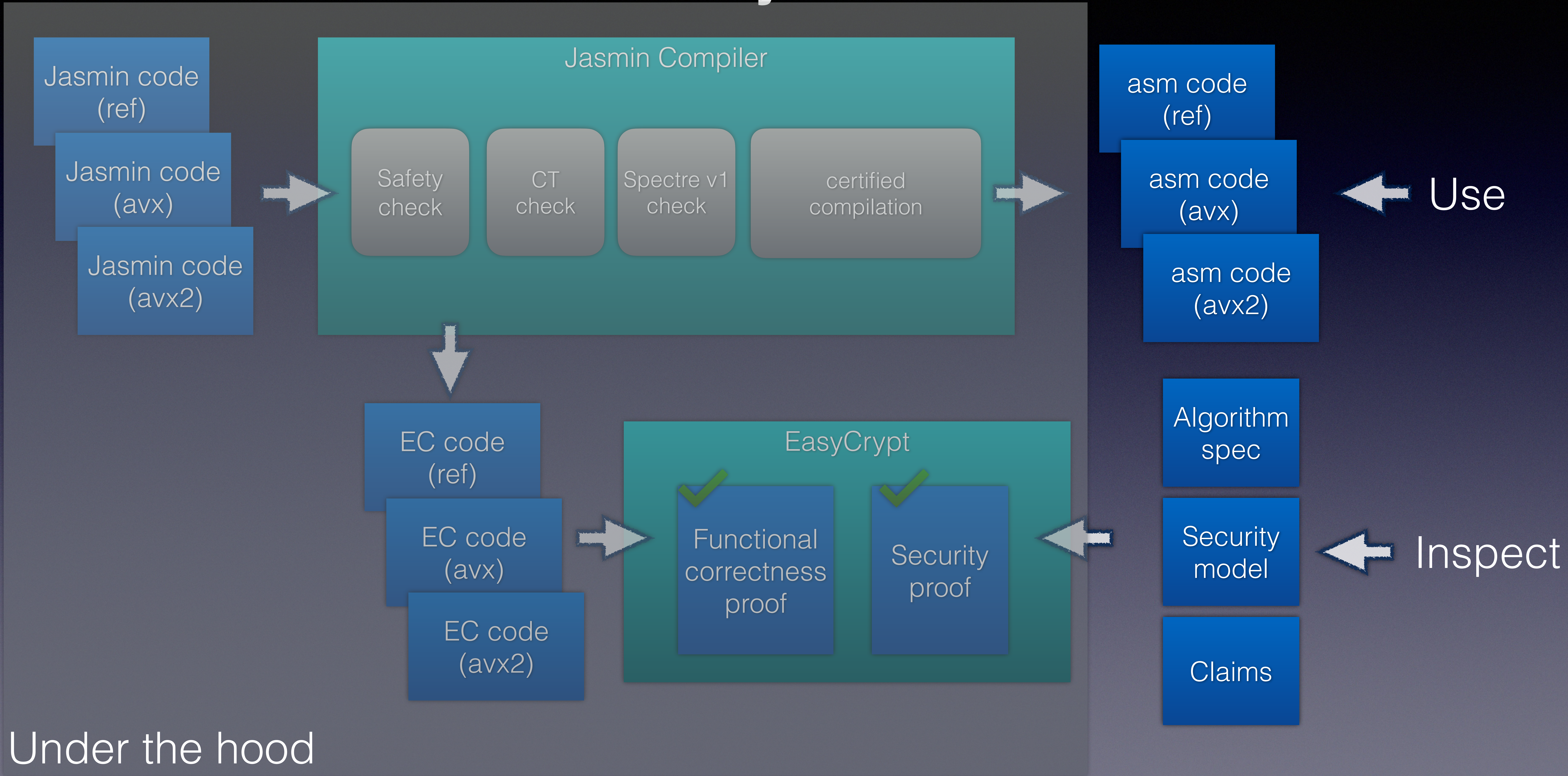
Interactively in a Zulip server

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)
  - 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 (Kyber, Dilithium)

libjade



Jasmin code (ref)

Jasmin code (avx)

Jasmin code (avx2)

Jasmin Compiler

Safety check

CT check

Spectre v1 check

certified compilation

asm code (ref)

asm code (avx)

asm code (avx2)

Use

EC code (ref)

EC code (avx)

EC code (avx2)

EasyCrypt

✓ Functional correctness proof

✓ Security proof

Algorithm spec

Security model

Claims

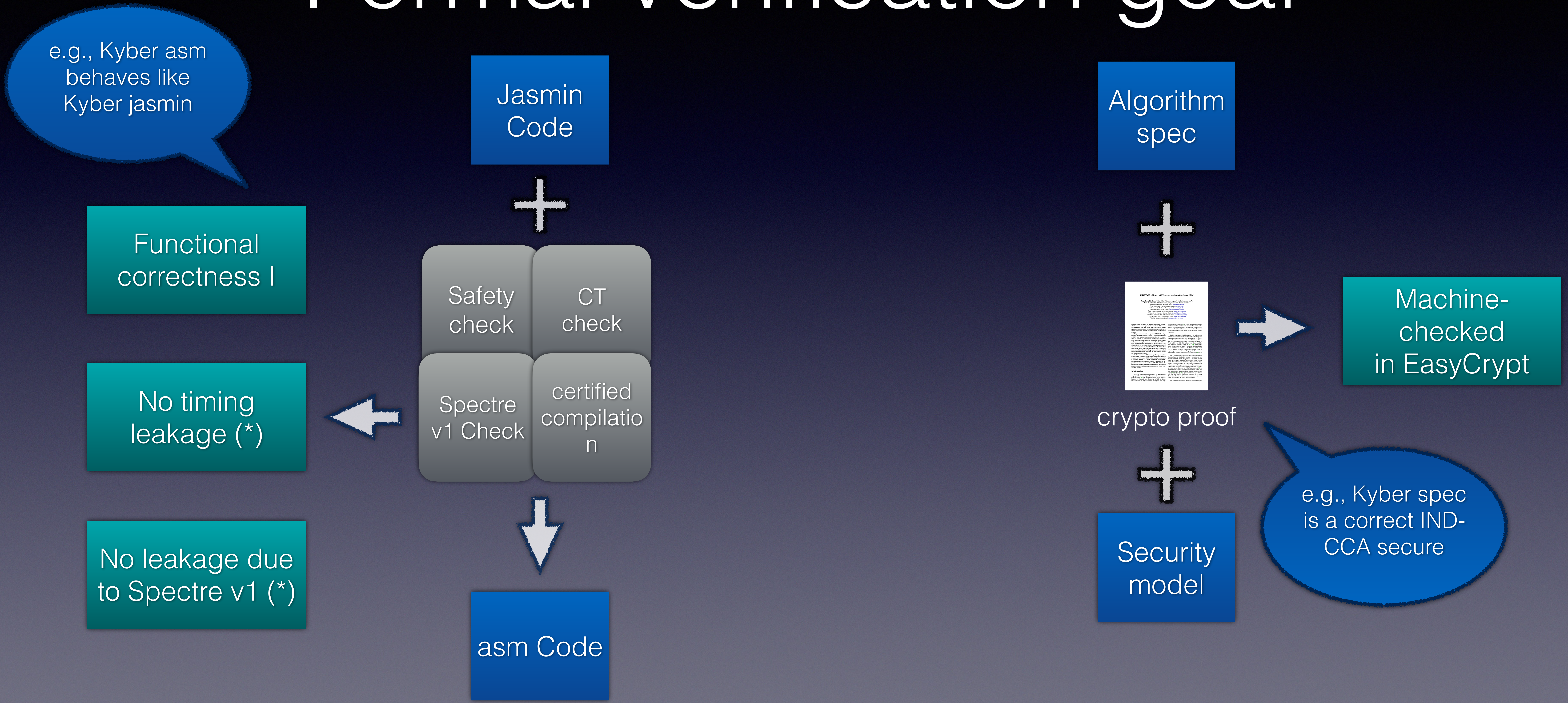
Inspect

Under the hood

Formal Verification Approach

- Formal verification goal
- Jasmin language and compiler
- EasyCrypt proof assistant

Formal verification goal



e.g., Kyber asm behaves like Kyber jasmin

Functional correctness I

No timing leakage (*)

No leakage due to Spectre v1 (*)

Jasmin Code

+

Safety check

CT check

Spectre v1 Check

certified compilation

asm Code

Algorithm spec

+

crypto proof

Machine-checked in EasyCrypt

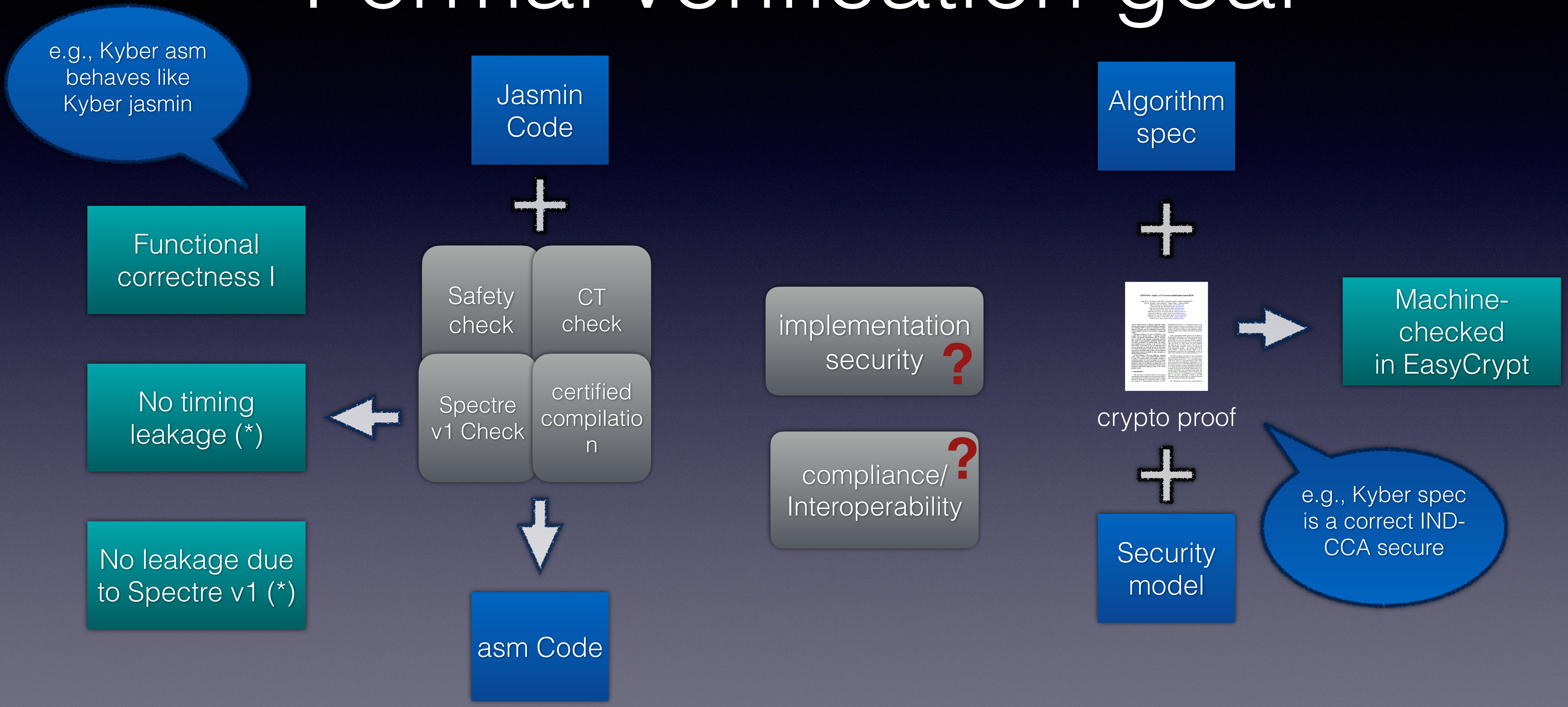
+

Security model

e.g., Kyber spec is a correct IND-CCA secure

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

Formal verification goal



e.g., Kyber asm behaves like Kyber jasmin

Functional correctness I

No timing leakage (*)

No leakage due to Spectre v1 (*)

Jasmin Code

+

Safety check

CT check

Spectre v1 Check

certified compilation

asm Code

implementation security ?

compliance/ Interoperability ?

Algorithm spec

+

crypto proof

+

Security model

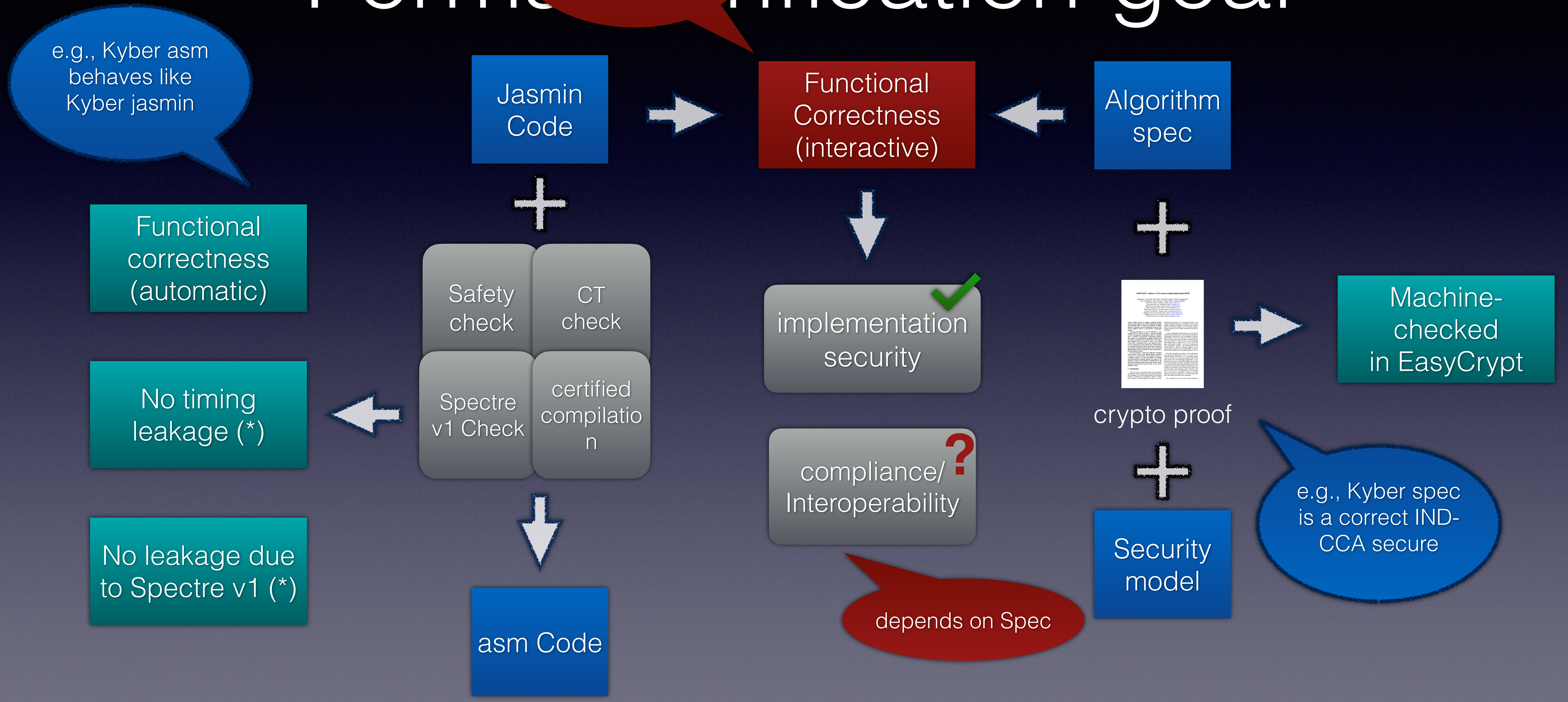
Machine-checked in EasyCrypt

e.g., Kyber spec is a correct IND-CCA secure

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

Forma ification goal

This talk!



e.g., Kyber asm behaves like Kyber jasmin

Jasmin Code

Functional Correctness (interactive)

Algorithm spec

Functional correctness (automatic)

Safety check

CT check

Spectre v1 Check

certified compilation

implementation security ✓

compliance/ Interoperability ?

depends on Spec

crypto proof

Machine-checked in EasyCrypt

No timing leakage (*)

No leakage due to Spectre v1 (*)

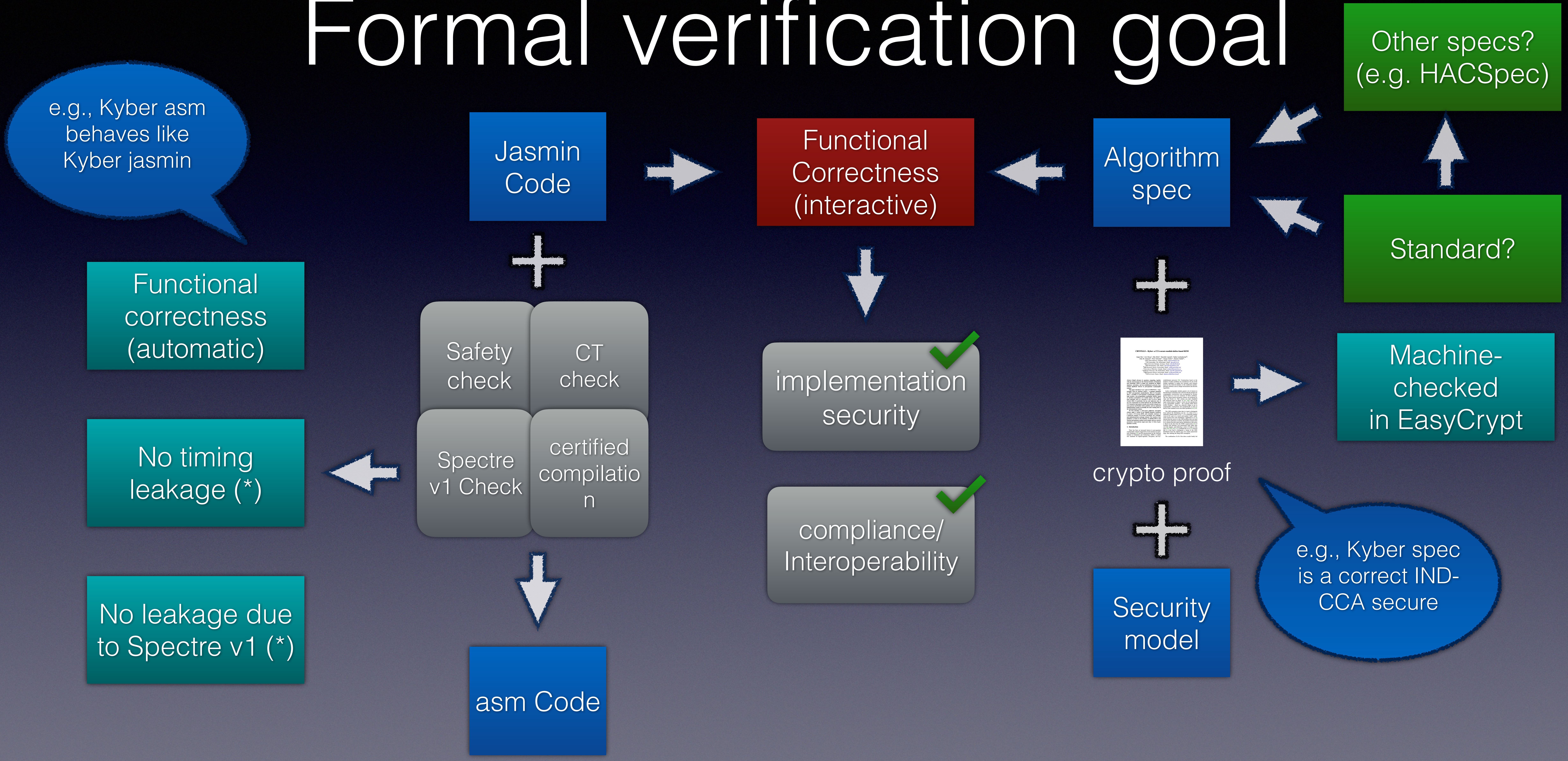
asm Code

Security model

e.g., Kyber spec is a correct IND-CCA secure

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

Formal verification goal



e.g., Kyber asm behaves like Kyber jasmin

Jasmin Code

Functional Correctness (interactive)

Algorithm spec

Other specs? (e.g. HACSpec)

Standard?

Functional correctness (automatic)

No timing leakage (*)

No leakage due to Spectre v1 (*)

Safety check

CT check

Spectre v1 Check

certified compilation

asm Code

implementation security ✓

compliance/ Interoperability ✓

crypto proof

Machine-checked in EasyCrypt

Security model

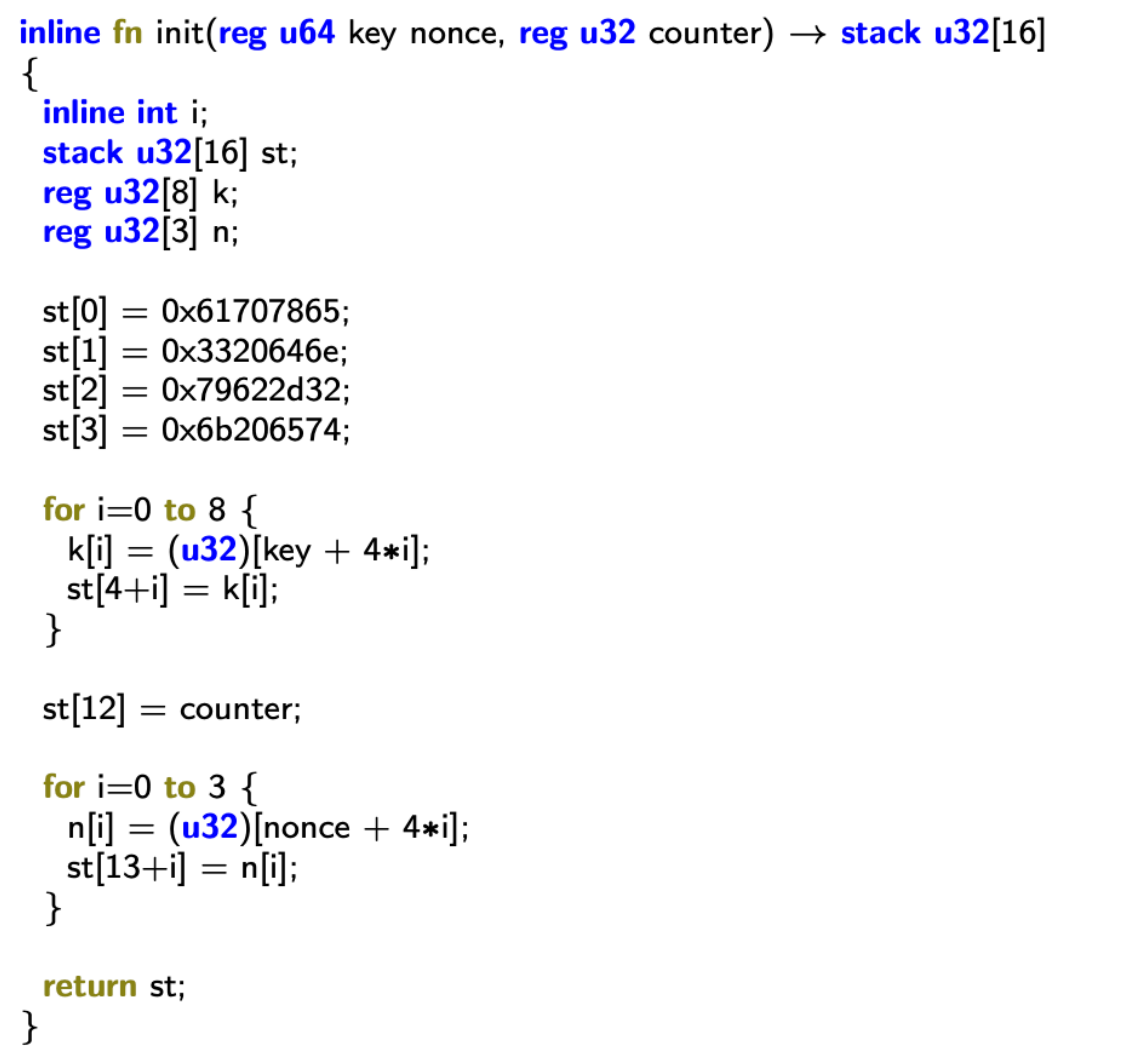
e.g., Kyber spec is a correct IND-CCA secure

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

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

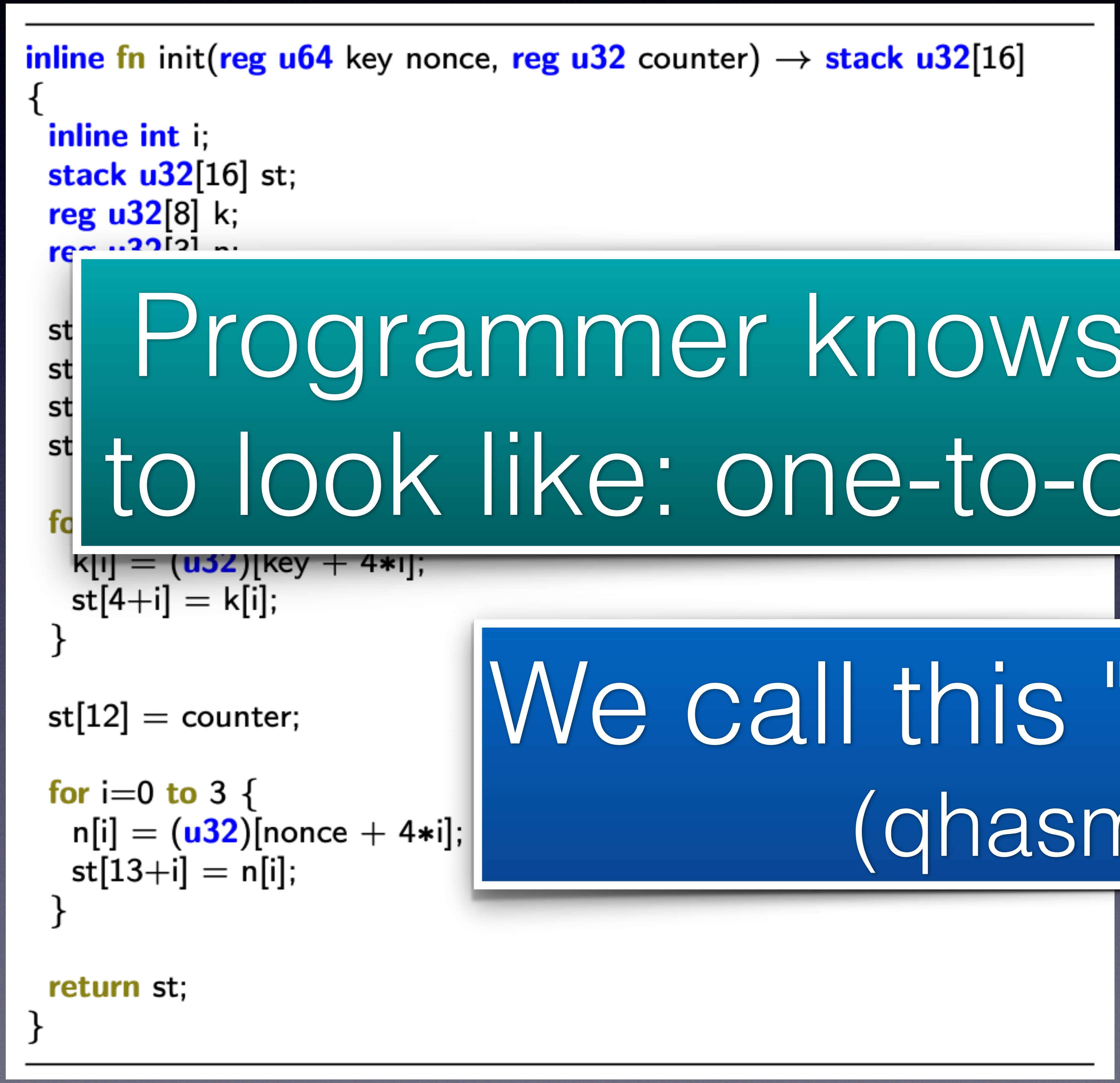
Jasmin: Zero cost abstractions



```
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;
  reg u32[3] n;

  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

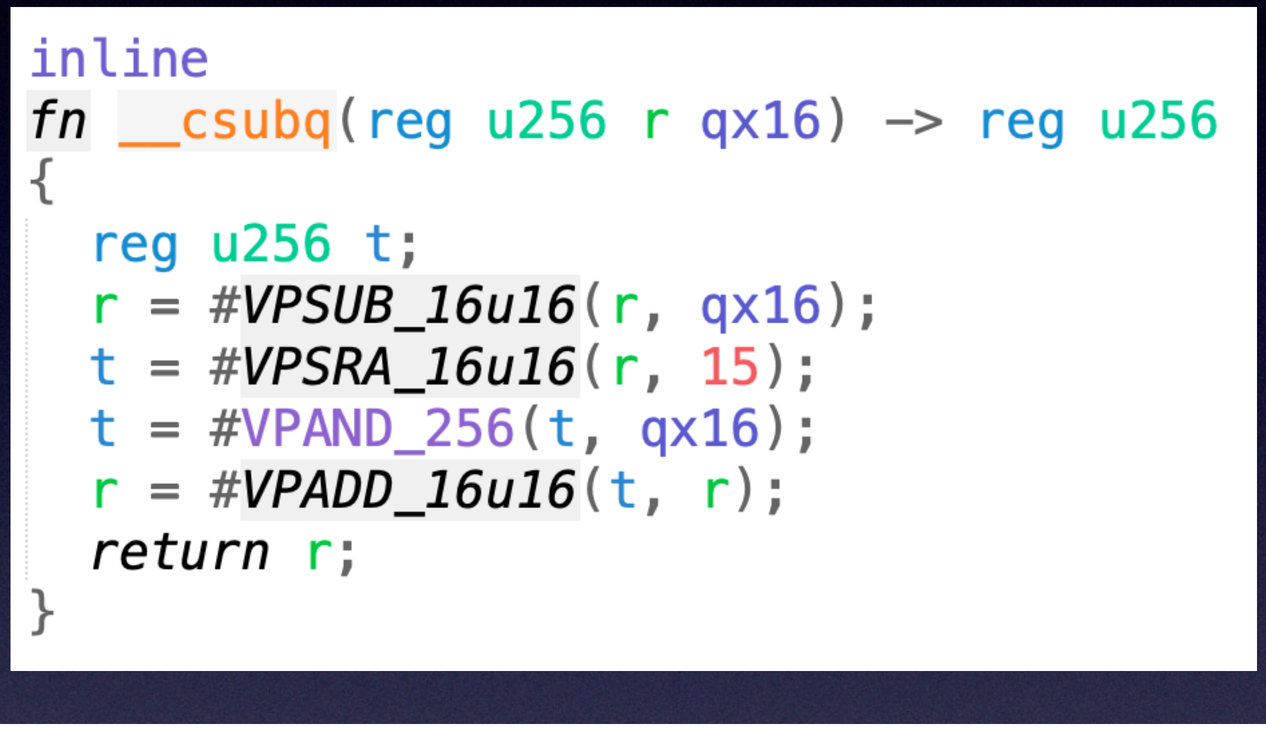
  es

  s
  - nice syntax and clever type checking

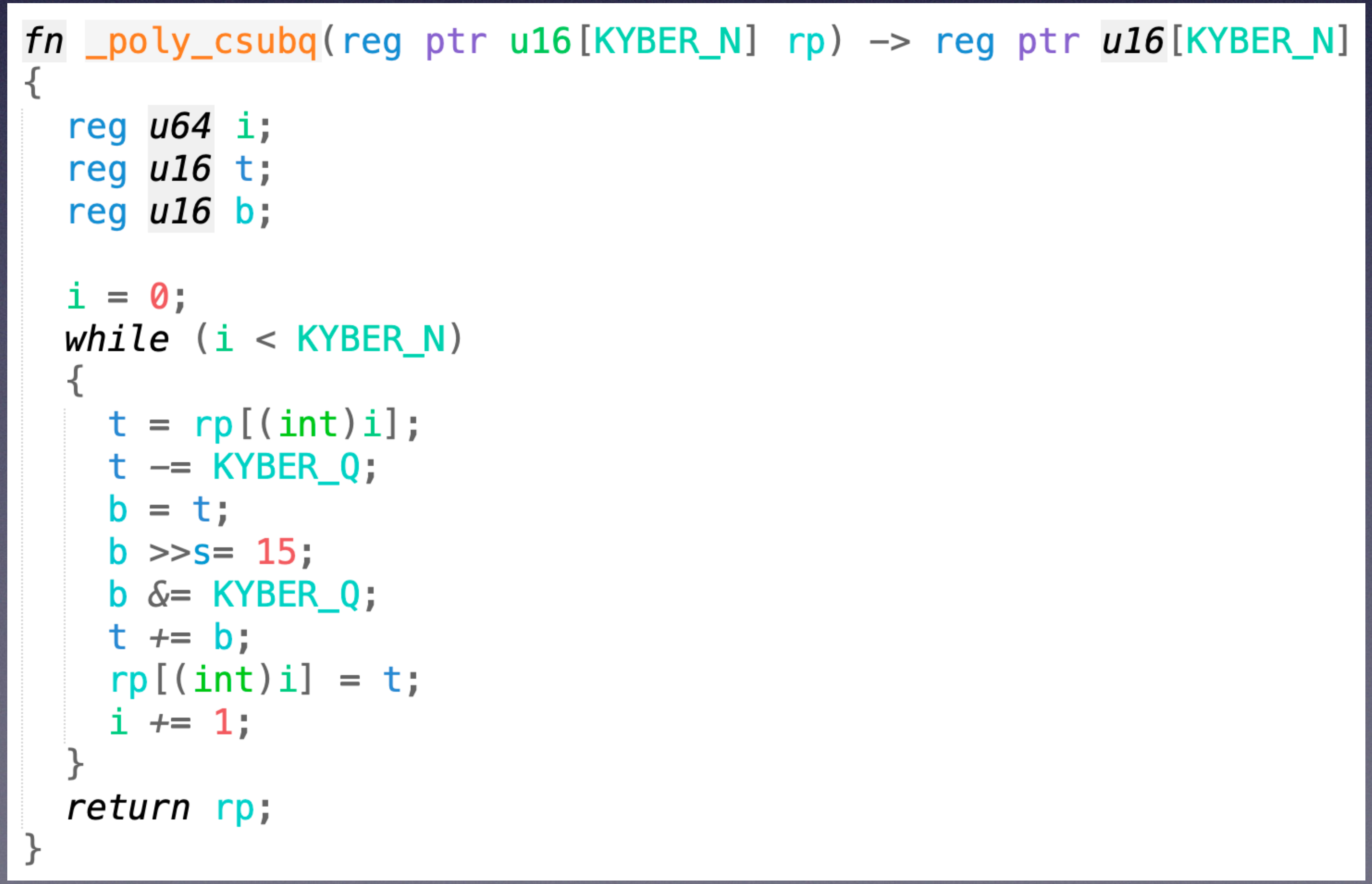
Programmer knows what assembly is going to look like: one-to-one instruction translation

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

Jasmin: per arch instruction set



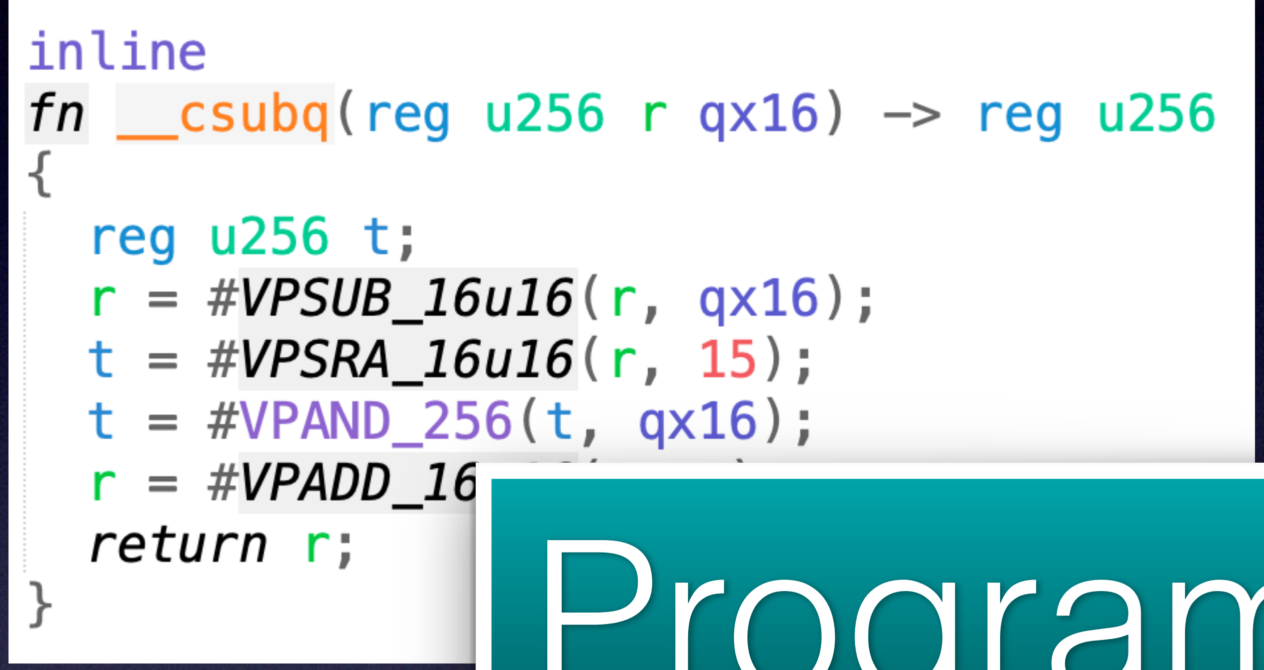
```
inline
fn __csubq(reg u256 r qx16) -> reg u256
{
  reg u256 t;
  r = #VPSUB_16u16(r, qx16);
  t = #VPSRA_16u16(r, 15);
  t = #VPAND_256(t, qx16);
  r = #VPADD_16u16(t, r);
  return r;
}
```



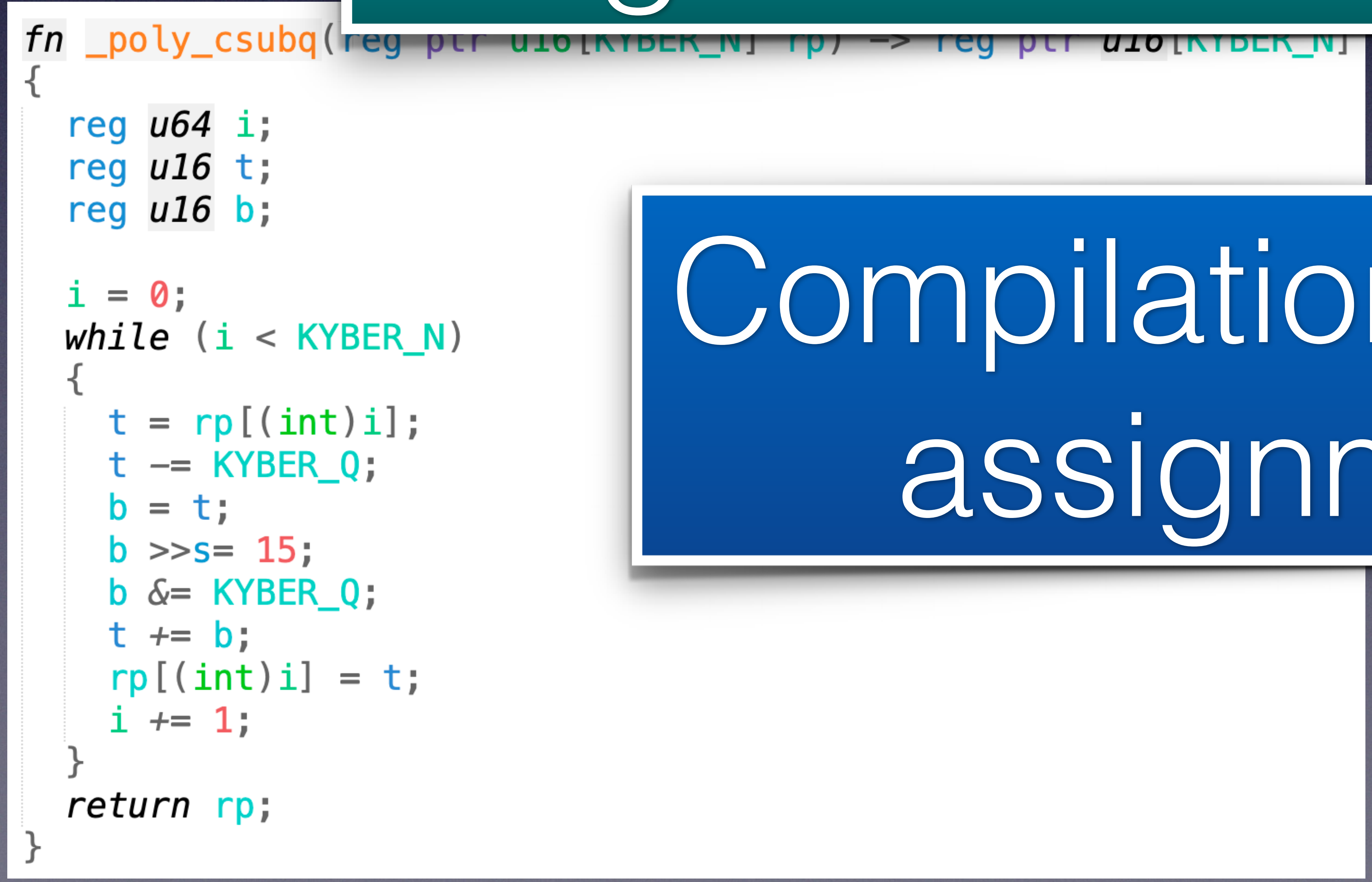
```
fn _poly_csubq(reg ptr u16[KYBER_N] rp) -> reg ptr u16[KYBER_N]
{
  reg u64 i;
  reg u16 t;
  reg u16 b;

  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;
}
```



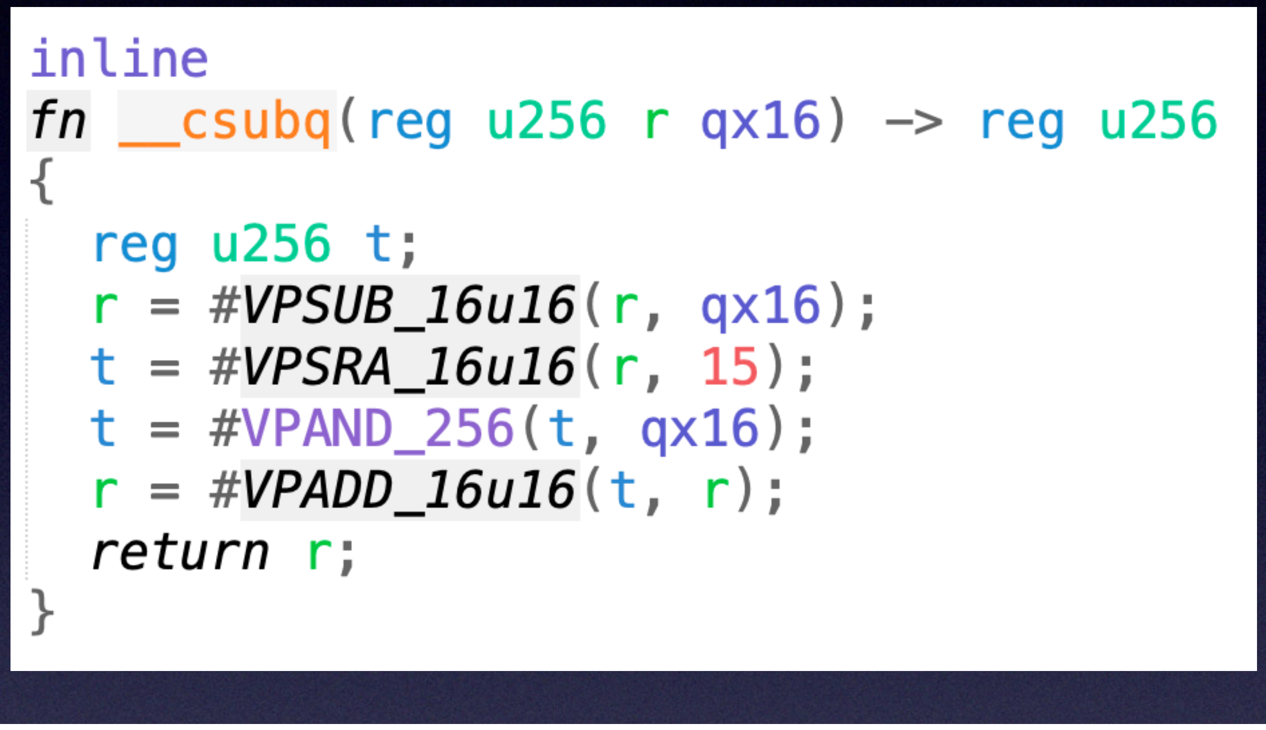
```
fn _poly_csubq(reg ptr u16[KYBER_N] rp) -> reg ptr u16[KYBER_N]
{
  reg u64 i;
  reg u16 t;
  reg u16 b;

  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

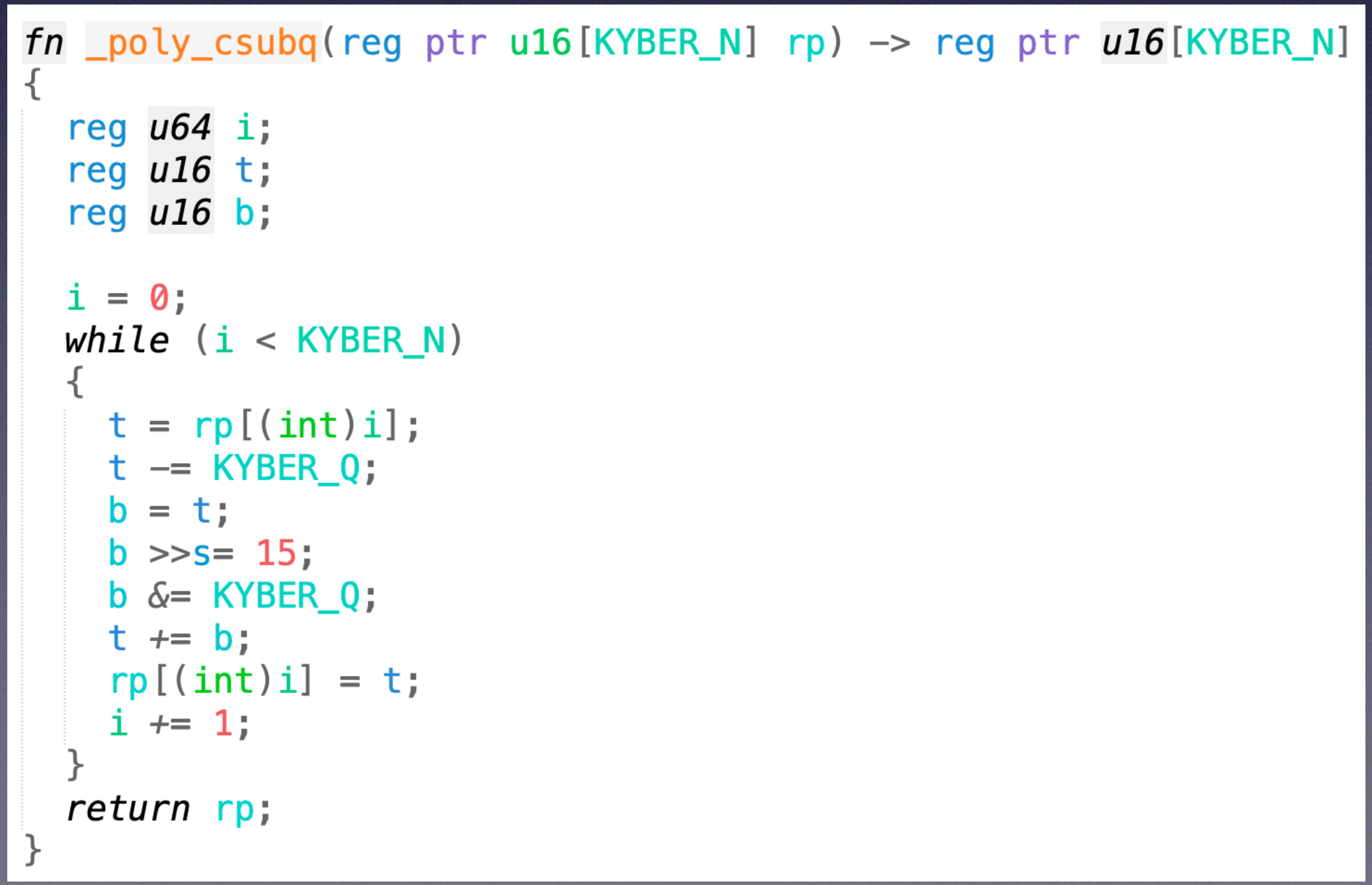
Programmer responsible for all spilling

Compilation breaks if register assignment not found.

Jasmin: per arch instruction set



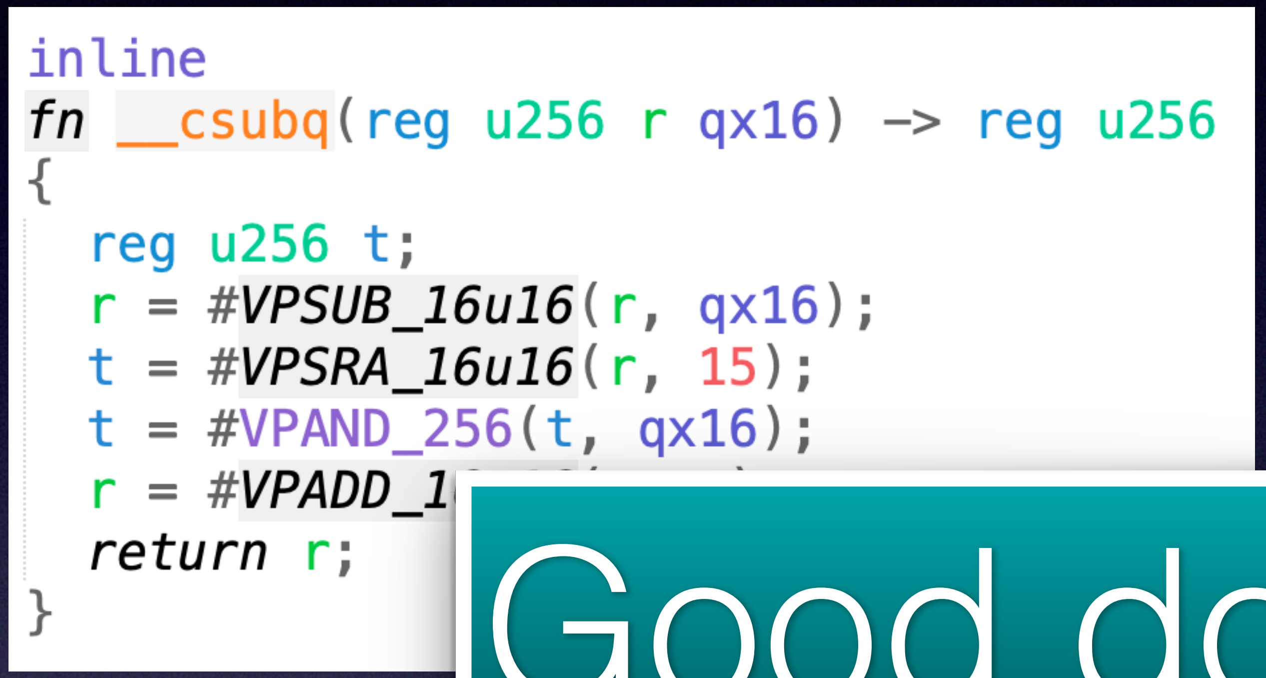
```
inline
fn __csubq(reg u256 r qx16) -> reg u256
{
  reg u256 t;
  r = #VPSUB_16u16(r, qx16);
  t = #VPSRA_16u16(r, 15);
  t = #VPAND_256(t, qx16);
  r = #VPADD_16u16(t, r);
  return r;
}
```



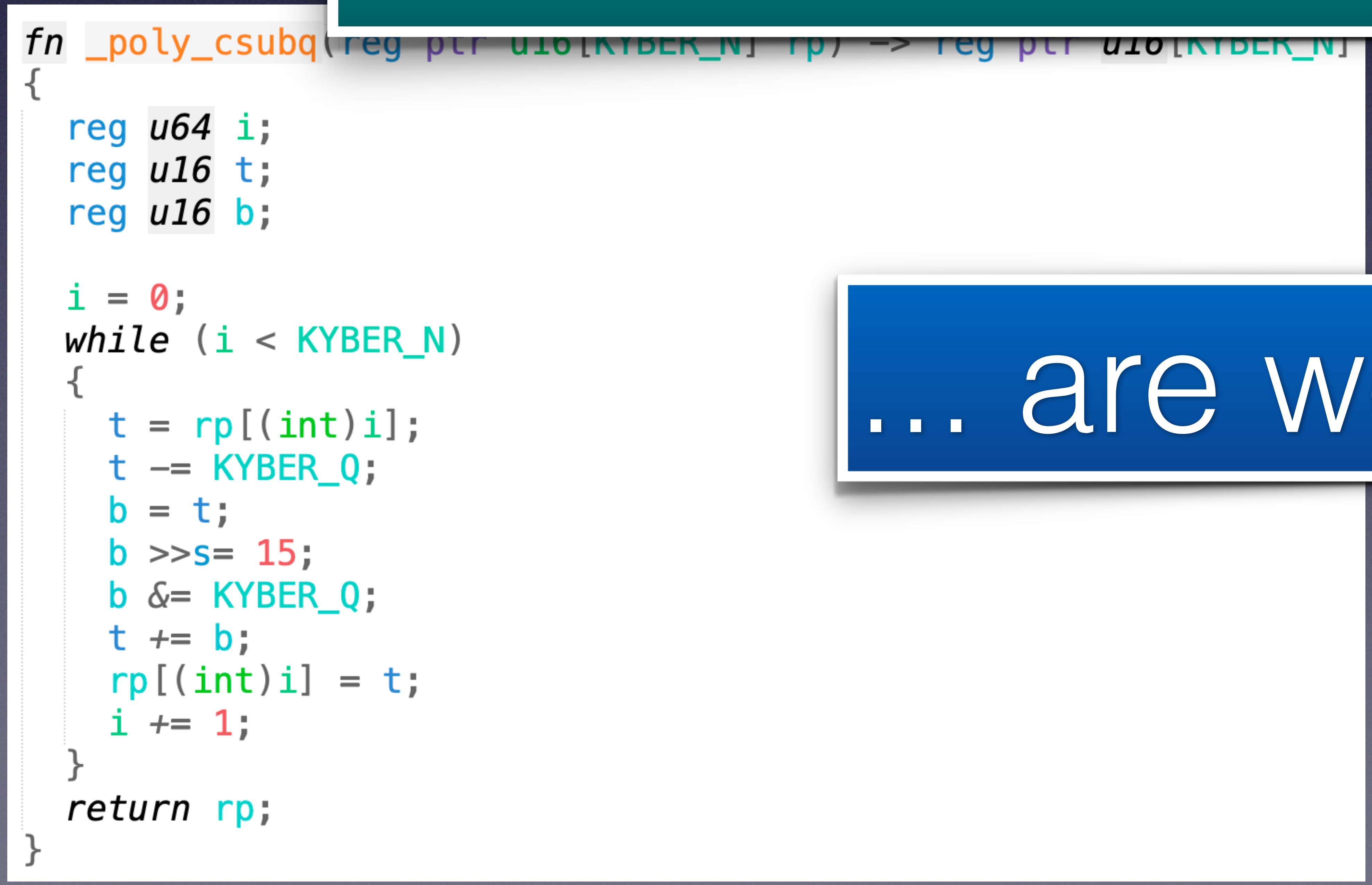
```
fn _poly_csubq(reg ptr u16[KYBER_N] rp) -> reg ptr u16[KYBER_N]
{
  reg u64 i;
  reg u16 t;
  reg u16 b;

  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;
}
```



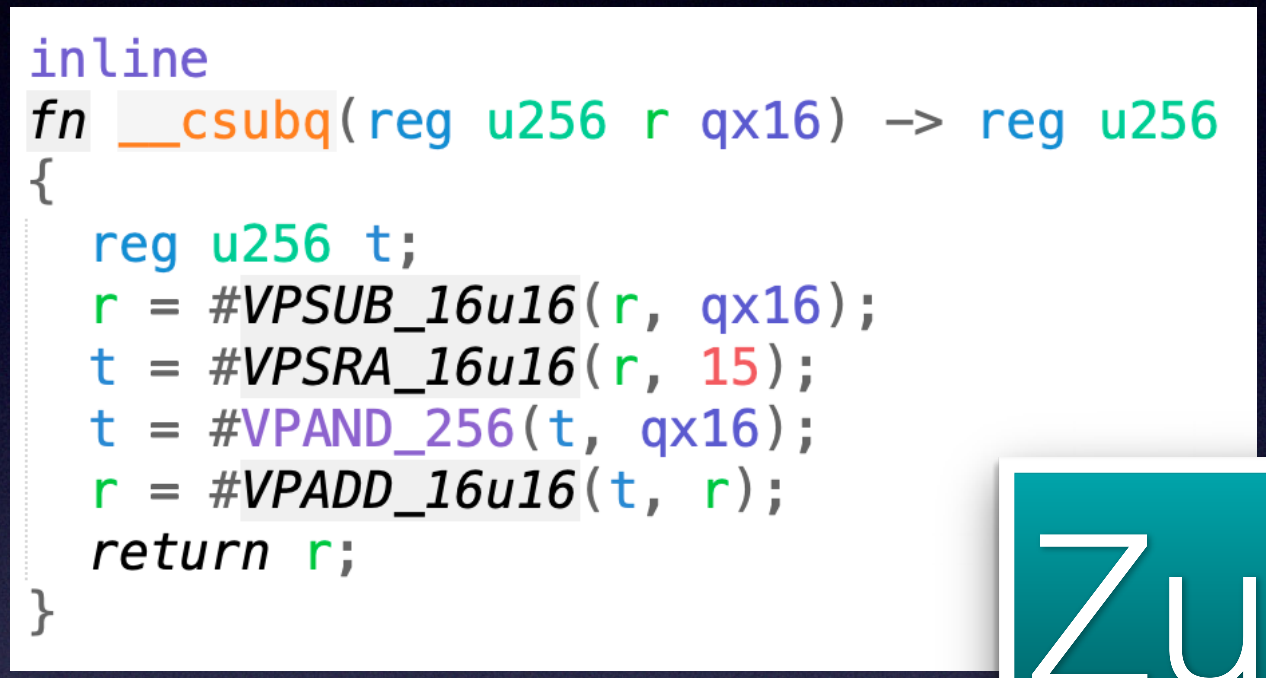
```
fn _poly_csubq(reg ptr u16[KYBER_N] rp) -> reg ptr u16[KYBER_N]
{
  reg u64 i;
  reg u16 t;
  reg u16 b;

  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

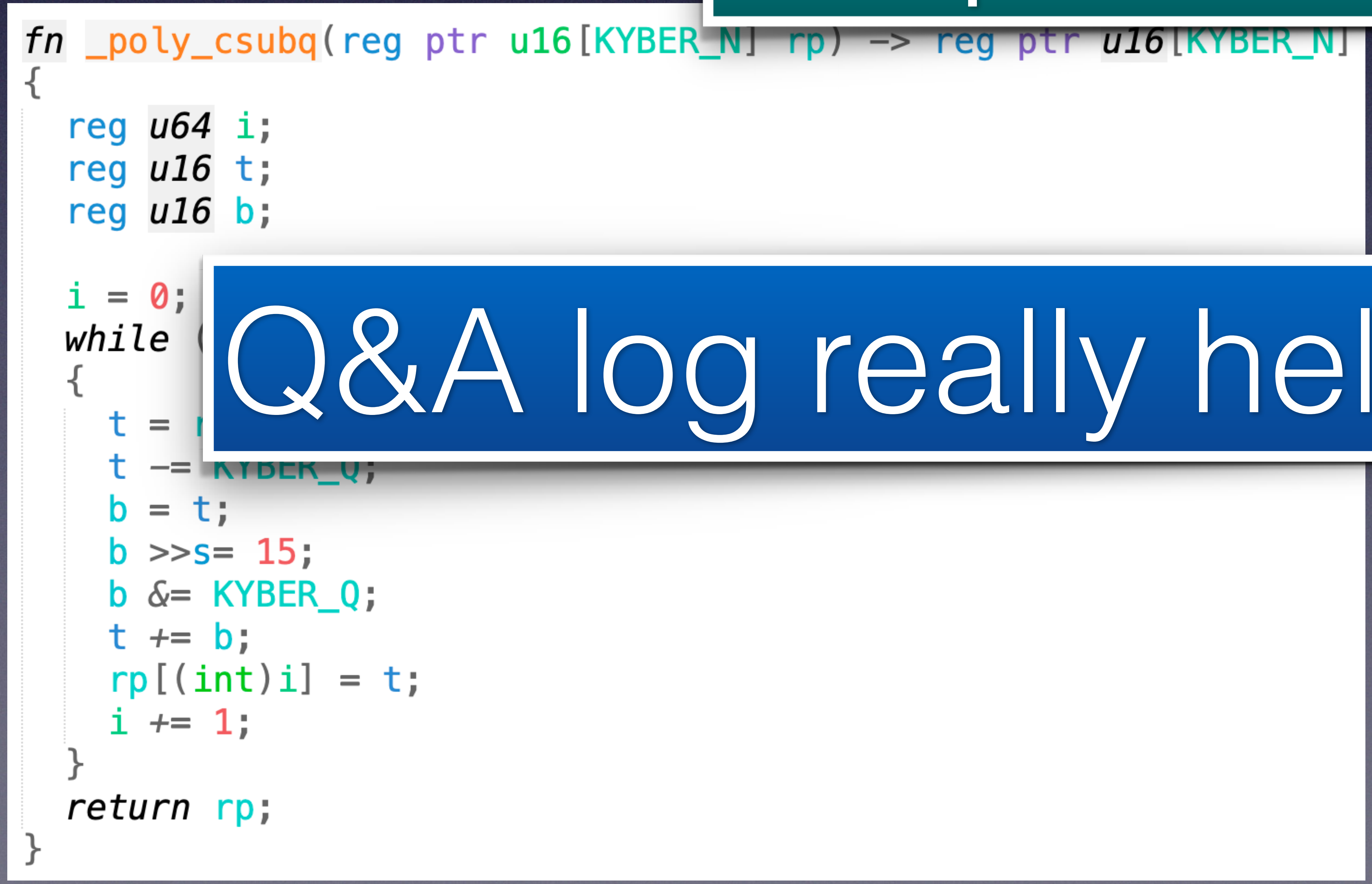
Good documentation and error msgs ...

... 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

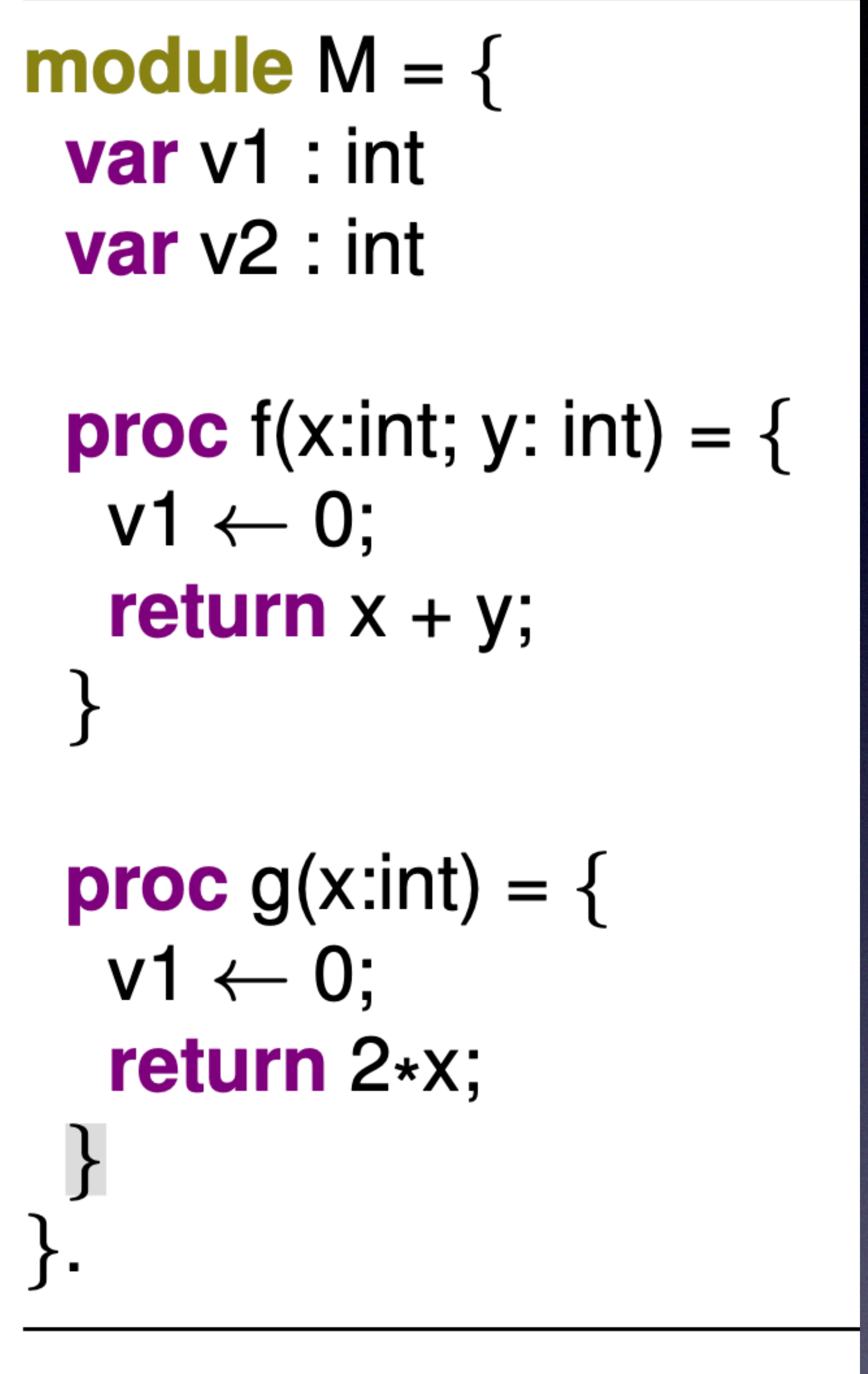
Zulip server is a good friend!

Q&A log really helps other users/developers.

EasyCrypt

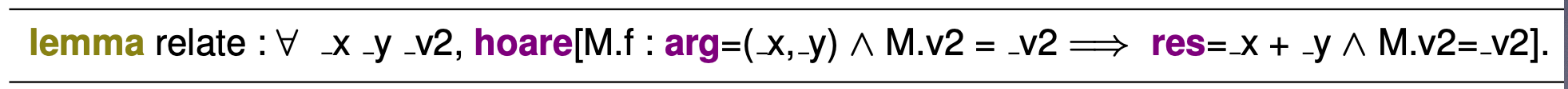
- Two languages: functional (define operators), imperative (implement algorithms)
- Logics to reason about properties of
  - real values (probabilities), distributions, etc.
  - functional programs (operators)
  - imperative programs (probabilistic Hoare logic or pHL)
  - relations between two imperative 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)

Hoare logic



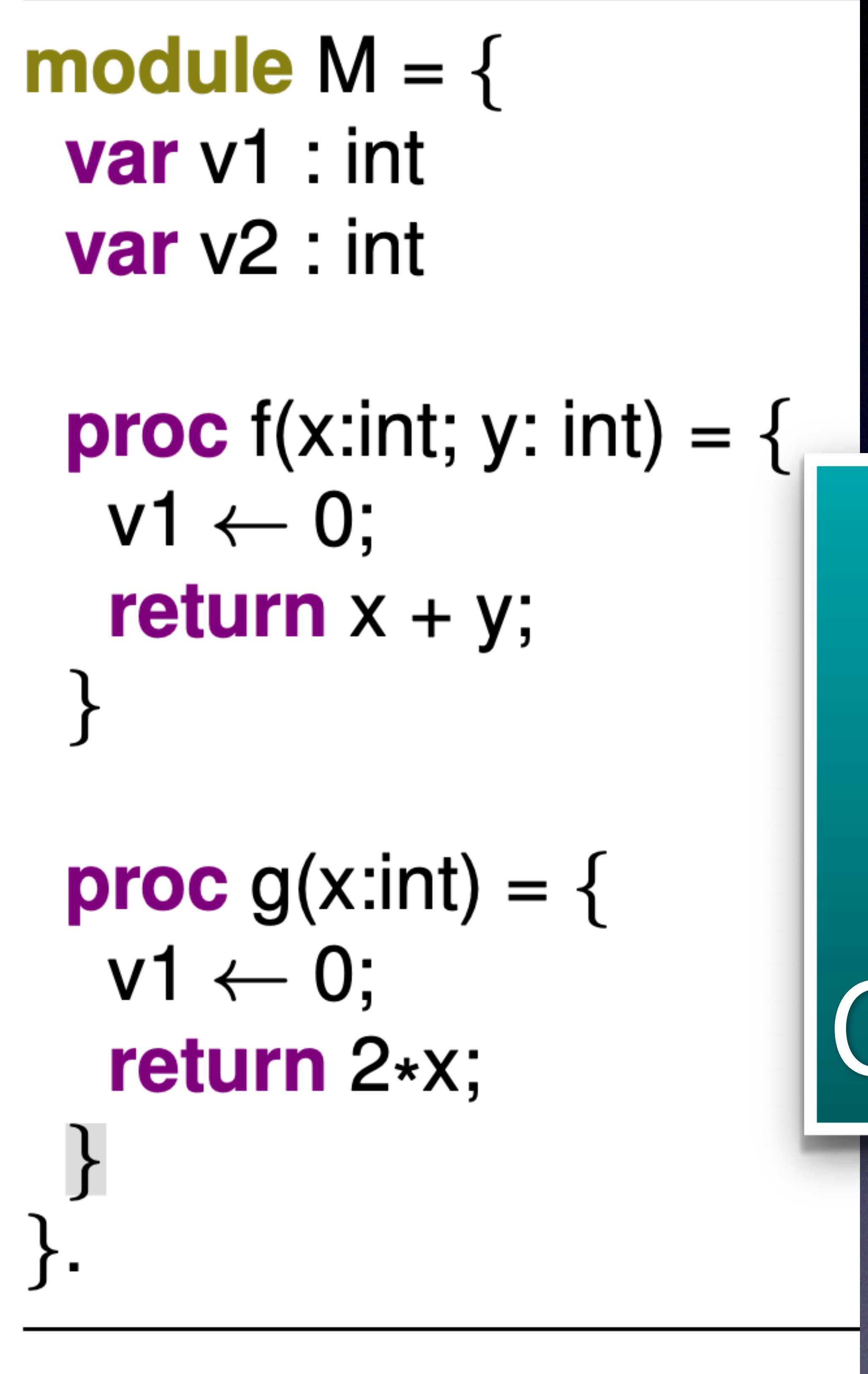
```
module M = {
  var v1 : int
  var v2 : int

  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].
```

Hoare logic



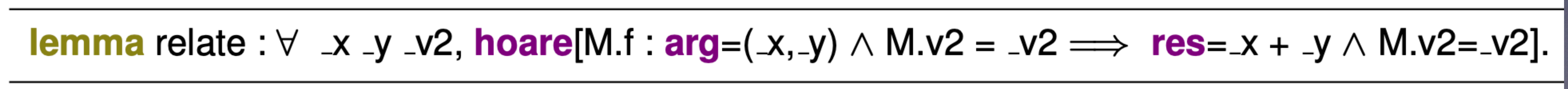
```
module M = {
  var v1 : int
  var v2 : int

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

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

In this work: prove that procedures implement convenient functional specs

predicates

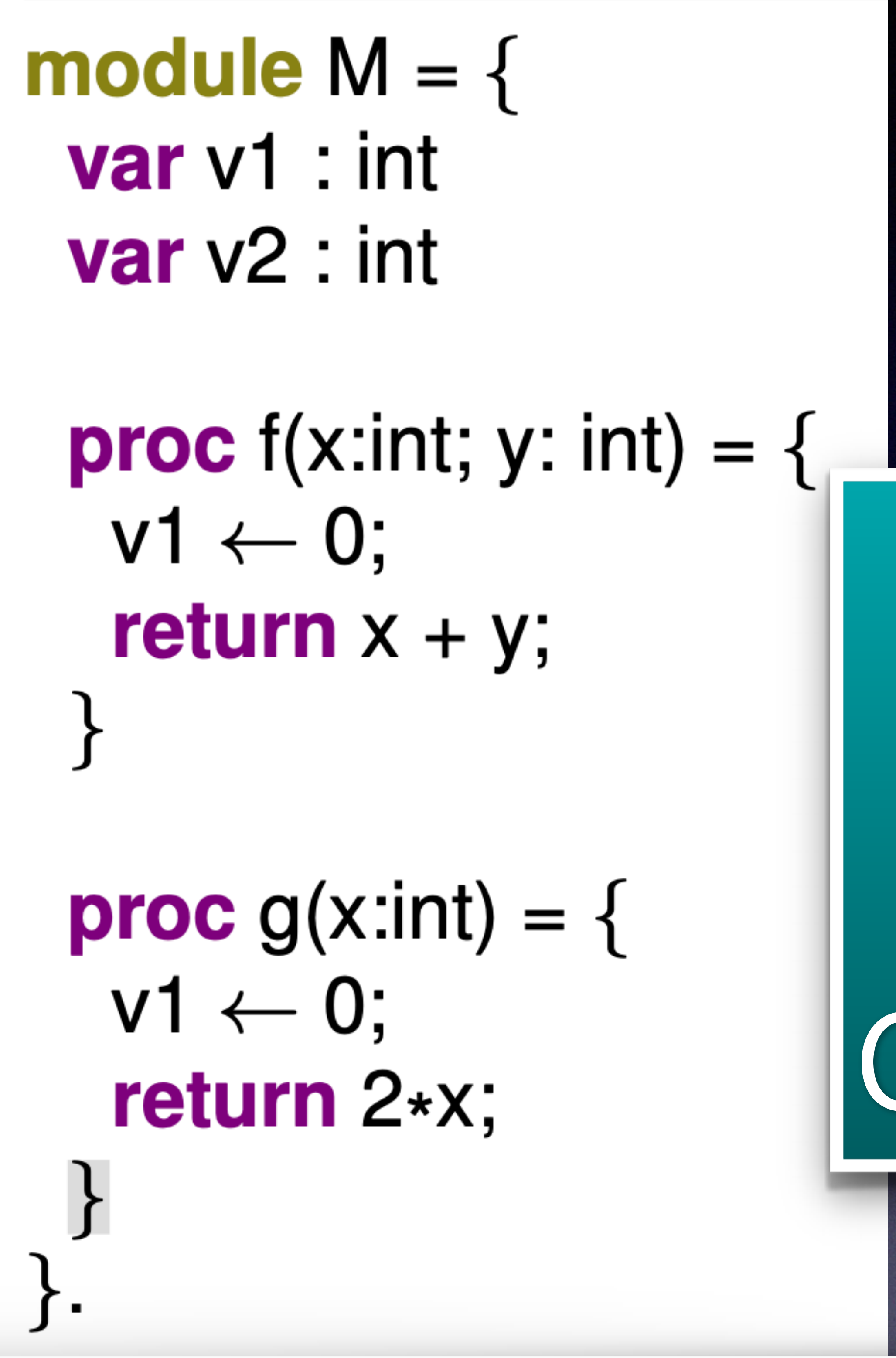
state

te



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

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;
  }
}.
```

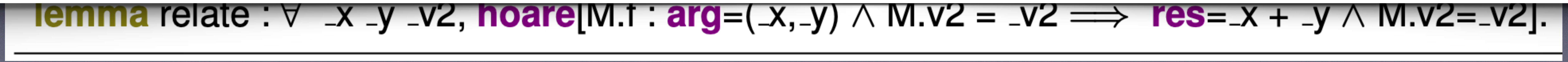
In this work: prove that procedures implement convenient functional specs

o predicates

state

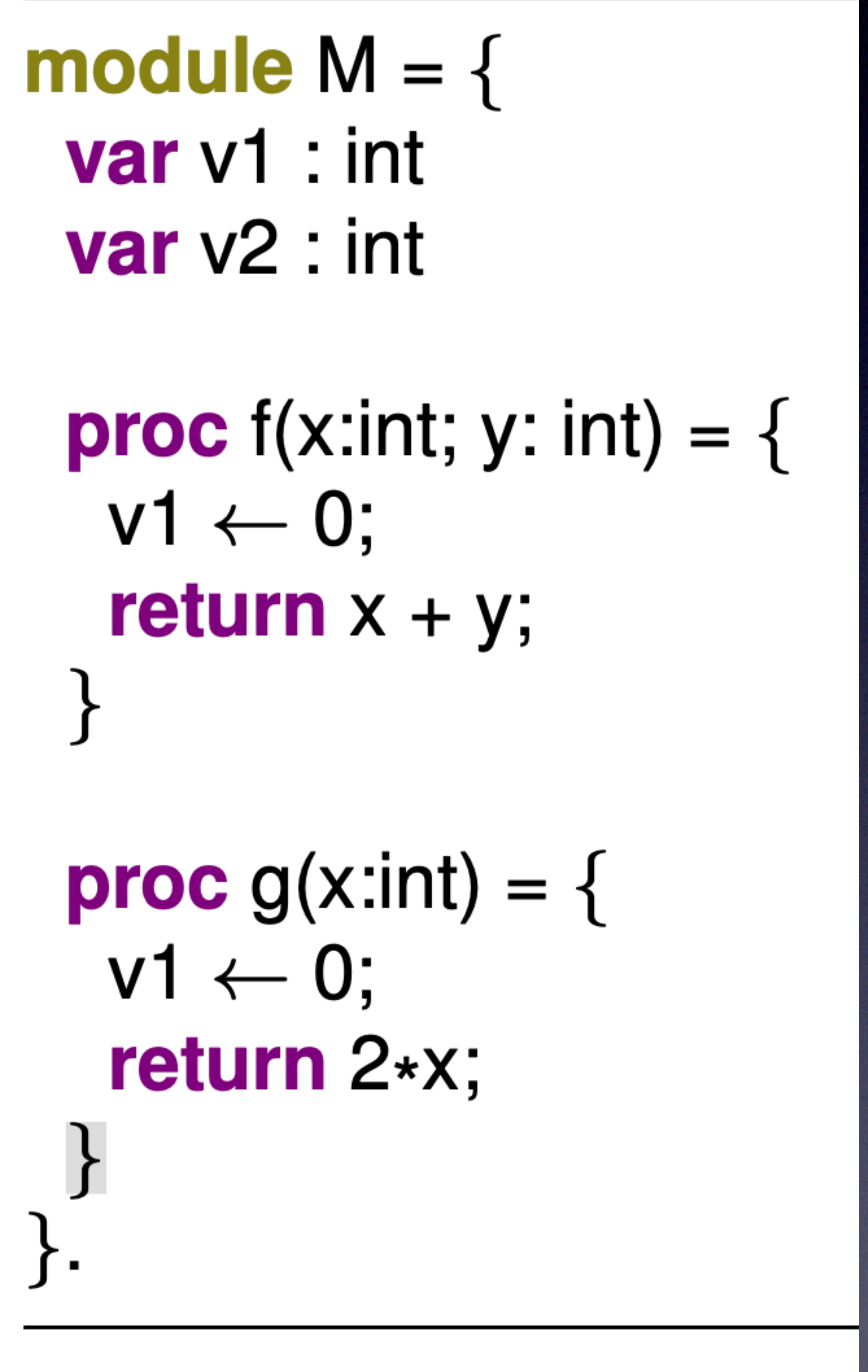
te

e.g., Jasmin code implements inner product correctly



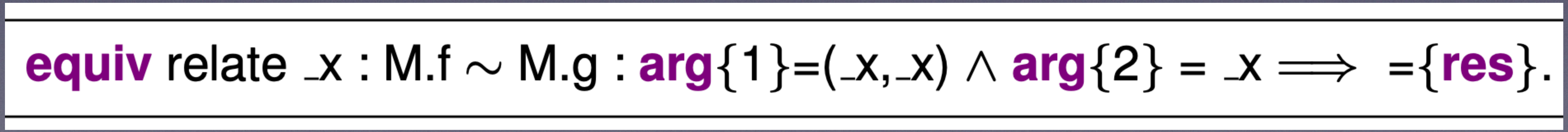
lemma relate : ∀ _x _y _v2, **hoare**[M.f : **arg**=(_x,_y) ∧ M.v2 = _v2 ⟹ **res**=_x + _y ∧ M.v2=_v2].

Relational Hoare logic



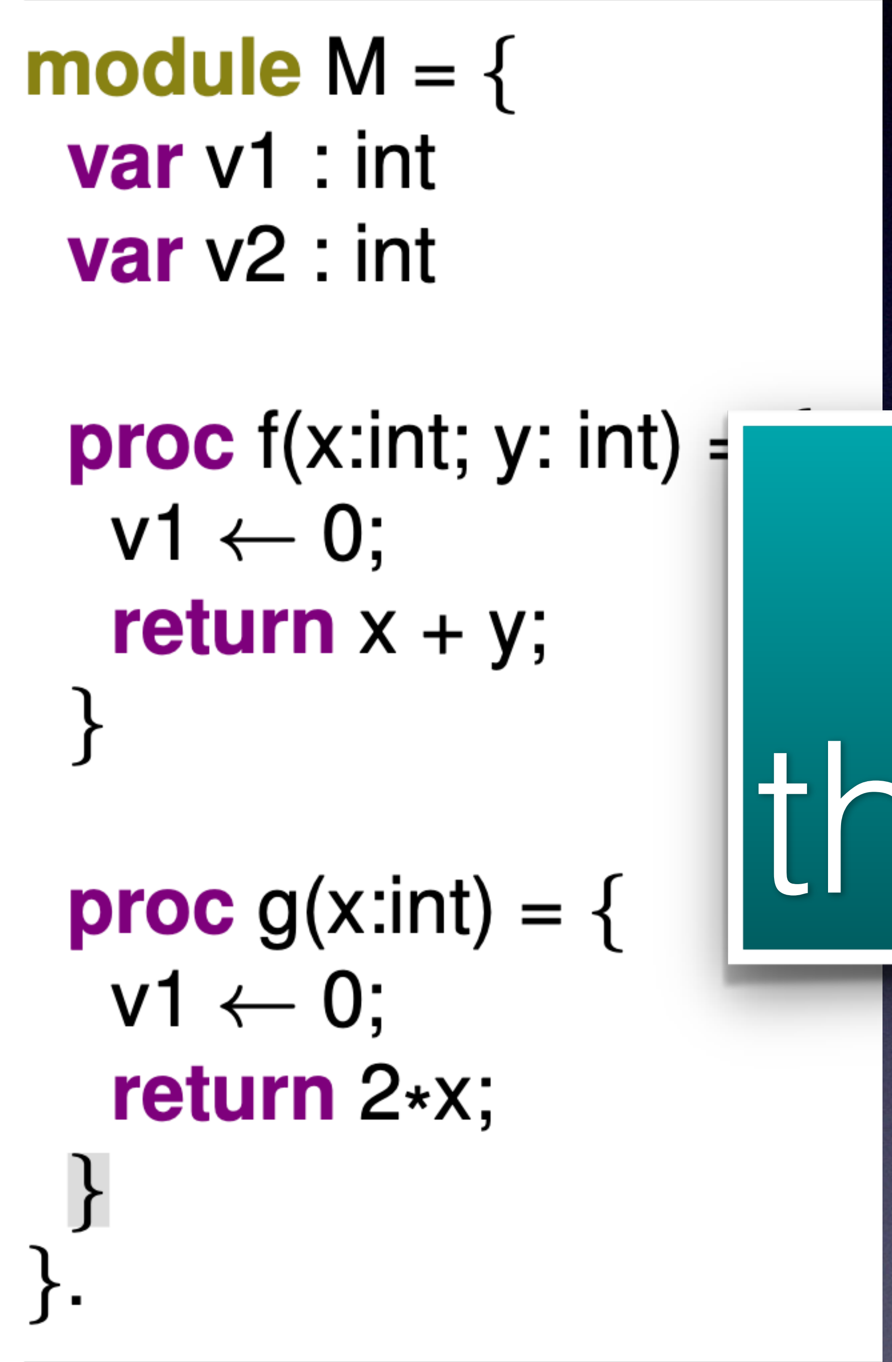
```
module M = {
  var v1 : int
  var v2 : int

  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}.
```

Relational Hoare logic



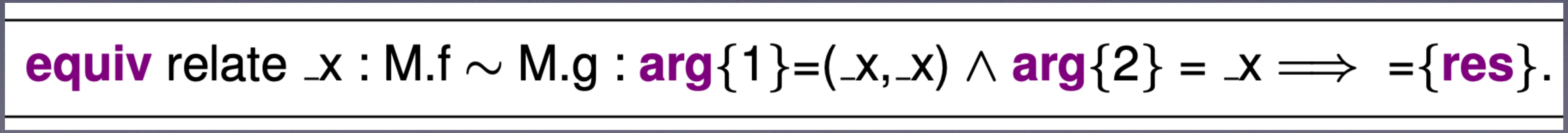
```
module M = {
  var v1 : int
  var v2 : int

  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

g states

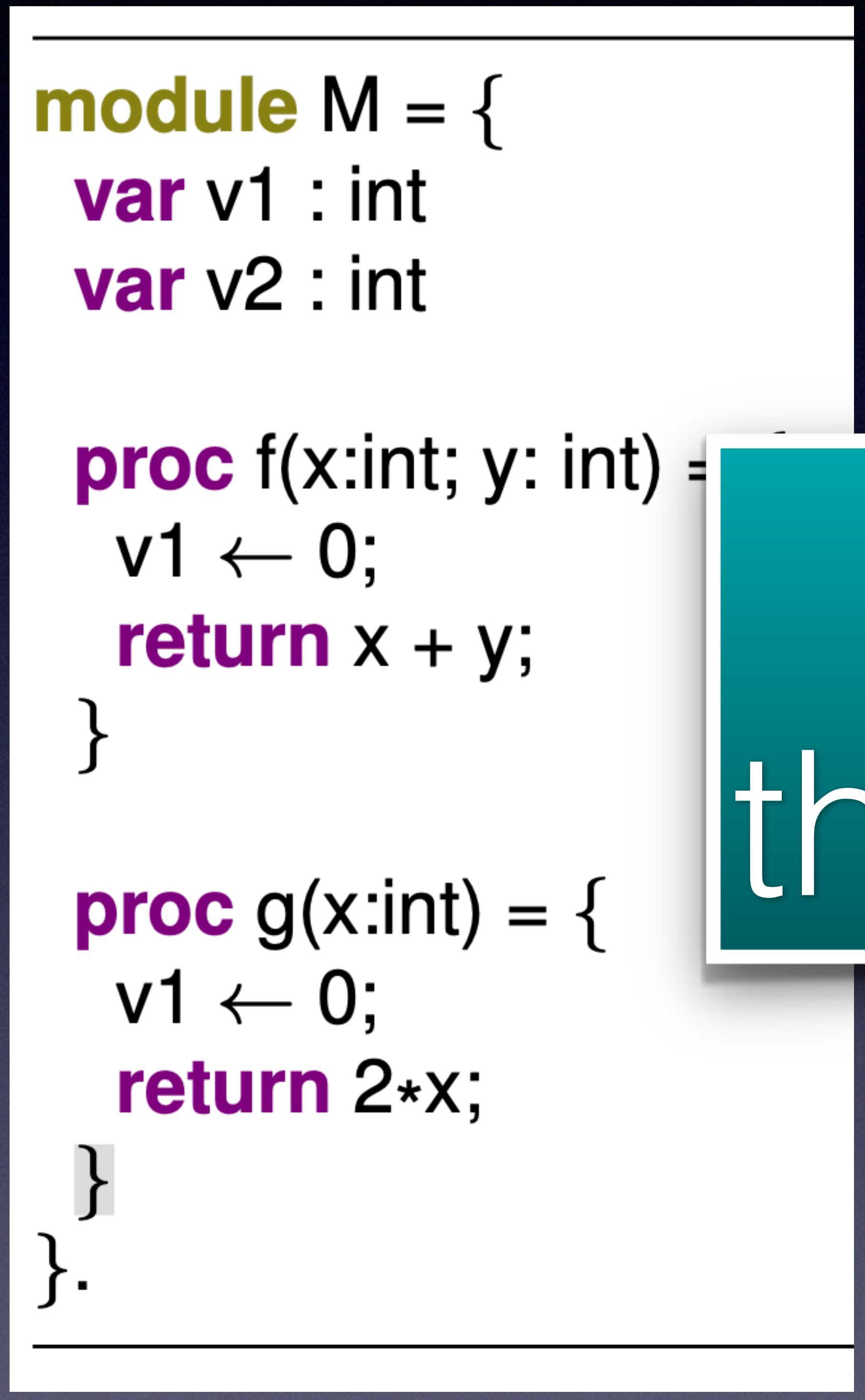
  - Postcondition: relation between final states

In this work: used to prove that two programs are equivalent.



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

Relational Hoare logic

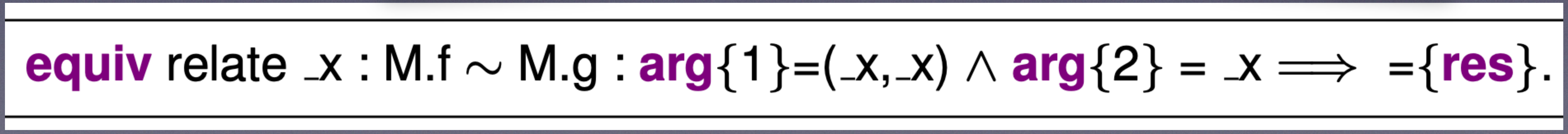


```
module M = {
  var v1 : int
  var v2 : int

  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
  - g states
  - Postcondition: relation between final states

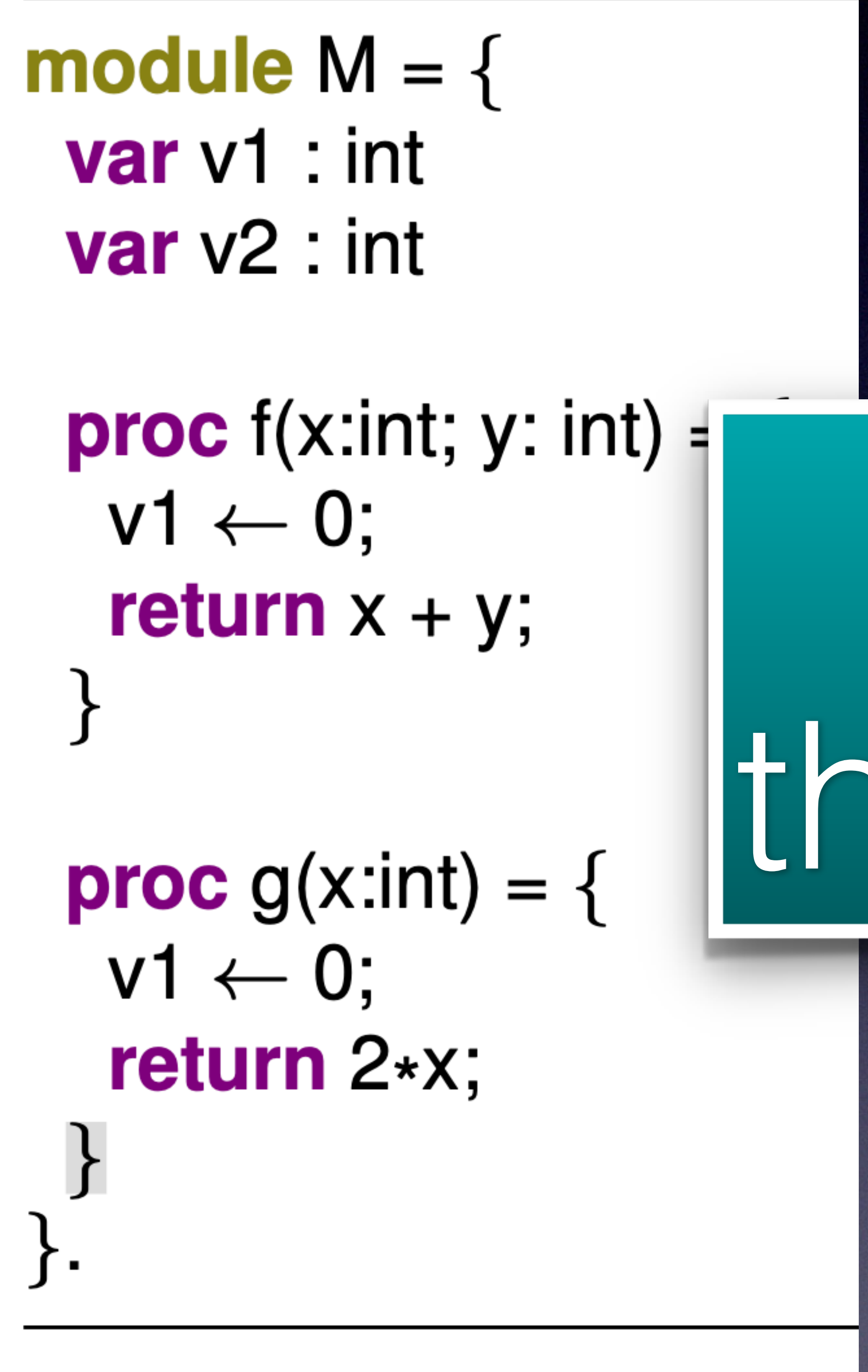
In this work: used to prove that two programs are equivalent.

spec vs implementation



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

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

g states

  - Postcondition: relation between final states

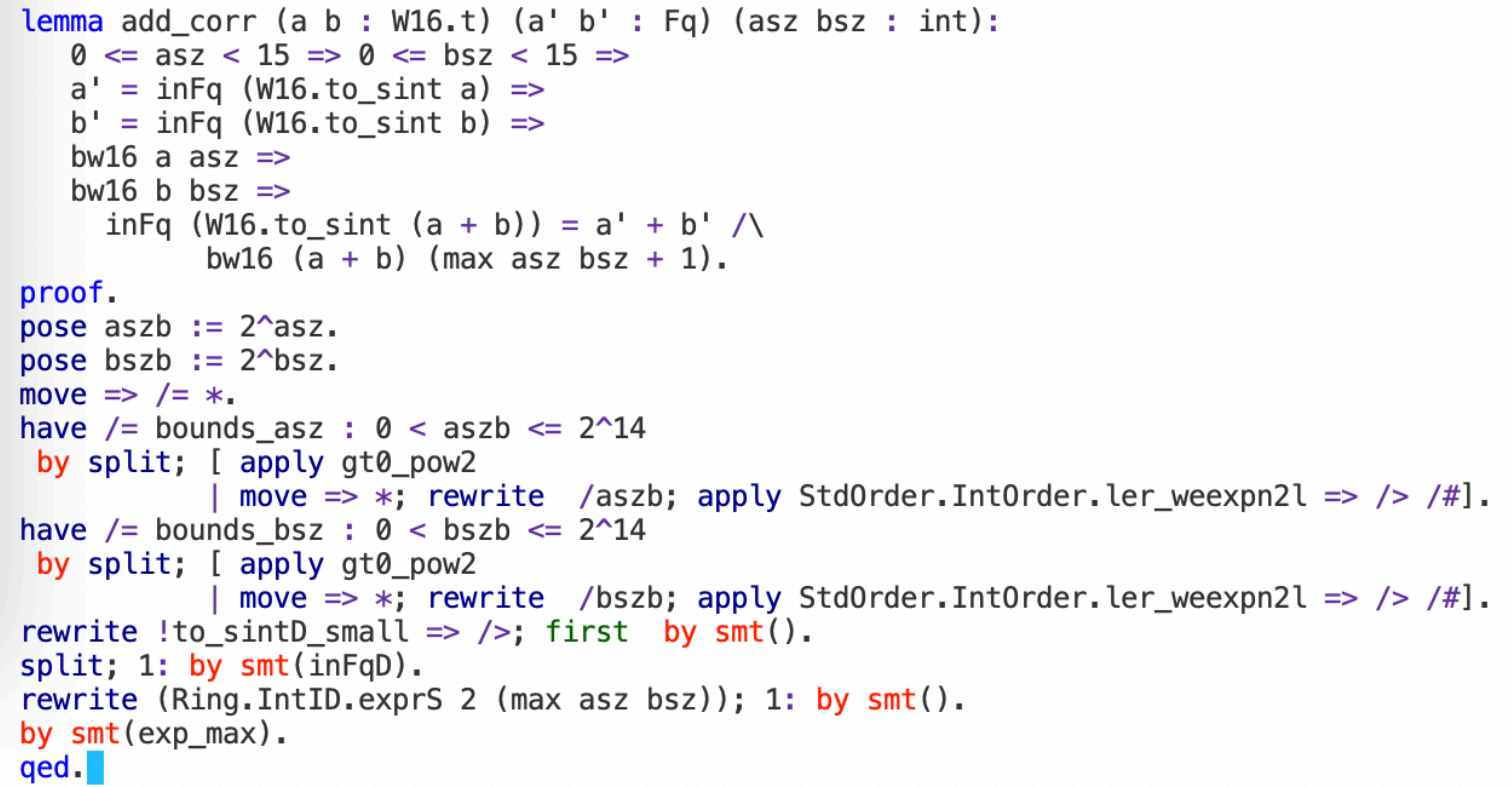
In this work: used to prove that two programs are equivalent.

implementation vs optimized implementation

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

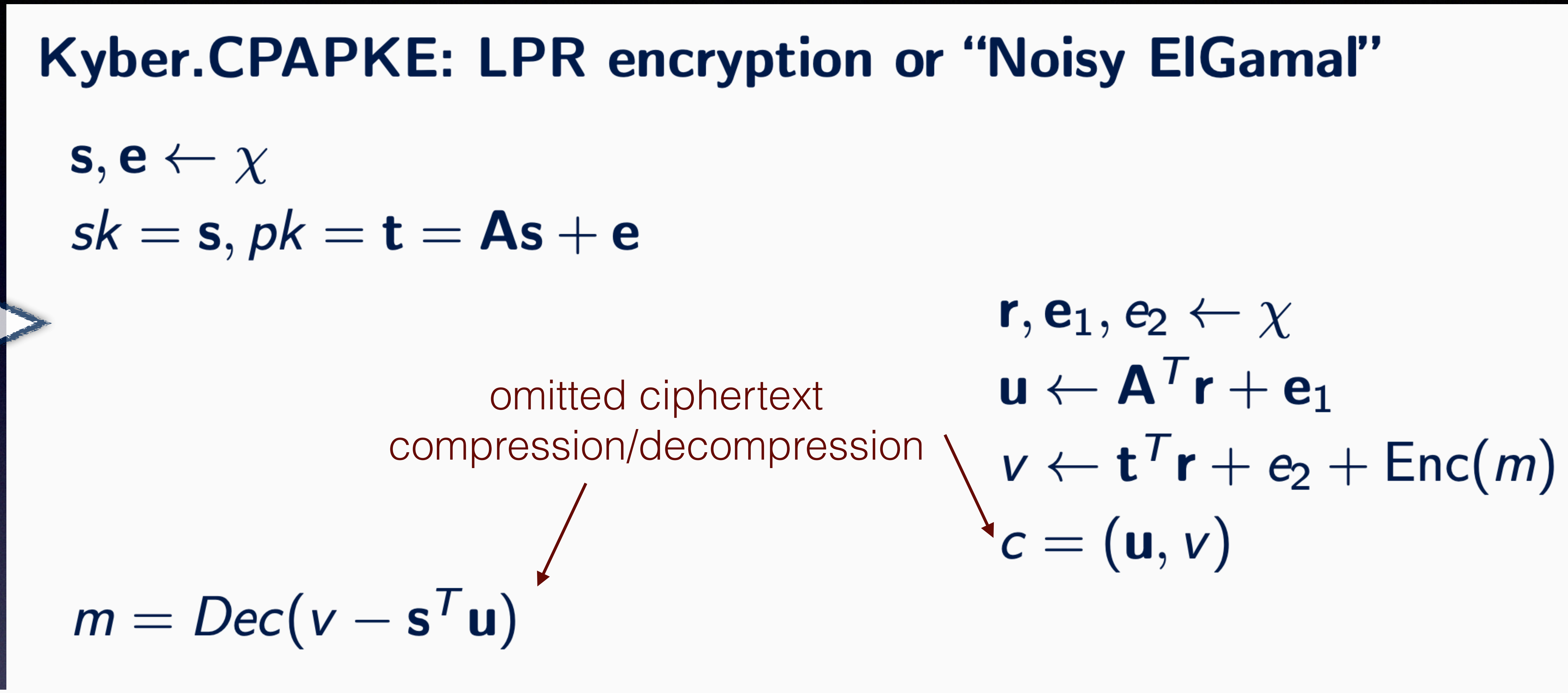
The Kyber Spec

- Kyber basics
- Specification goals
- Snippets/examples

Kyber Basics

q = 3329 is a prime
Fq: field, integers modulo q, type of coefficients
Rq: ring of polynomials modulo ($X^{256}+1$) over Fq
Bold lower caps: col vectors of size k over Rq
Bold upper caps: k x k matrix over Rq

s, **e**, **r**, $\mathbf{e_1}$, e_2 small norm: each coeff. Binomial distr.
A coeffs. sampled uniformly from Fq
Multiplications in Rq done in NTT domain
Enc/Dec: encoding and decoding operations



Kyber.CPAPKE: LPR encryption or “Noisy ElGamal”

$\mathbf{s}, \mathbf{e} \leftarrow \chi$

$sk = \mathbf{s}, pk = \mathbf{t} = \mathbf{A}\mathbf{s} + \mathbf{e}$

$\mathbf{r}, \mathbf{e}_1, e_2 \leftarrow \chi$

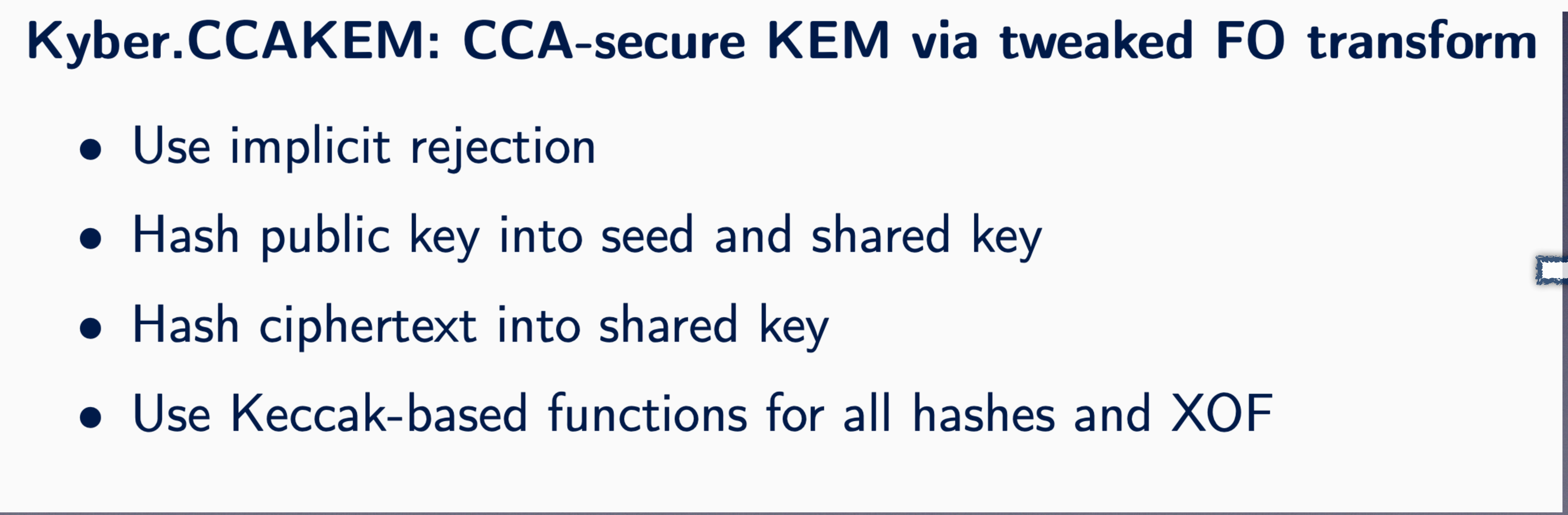
$\mathbf{u} \leftarrow \mathbf{A}^T\mathbf{r} + \mathbf{e}_1$

$v \leftarrow \mathbf{t}^T\mathbf{r} + e_2 + \mathrm{Enc}(m)$

$c = (\mathbf{u}, v)$

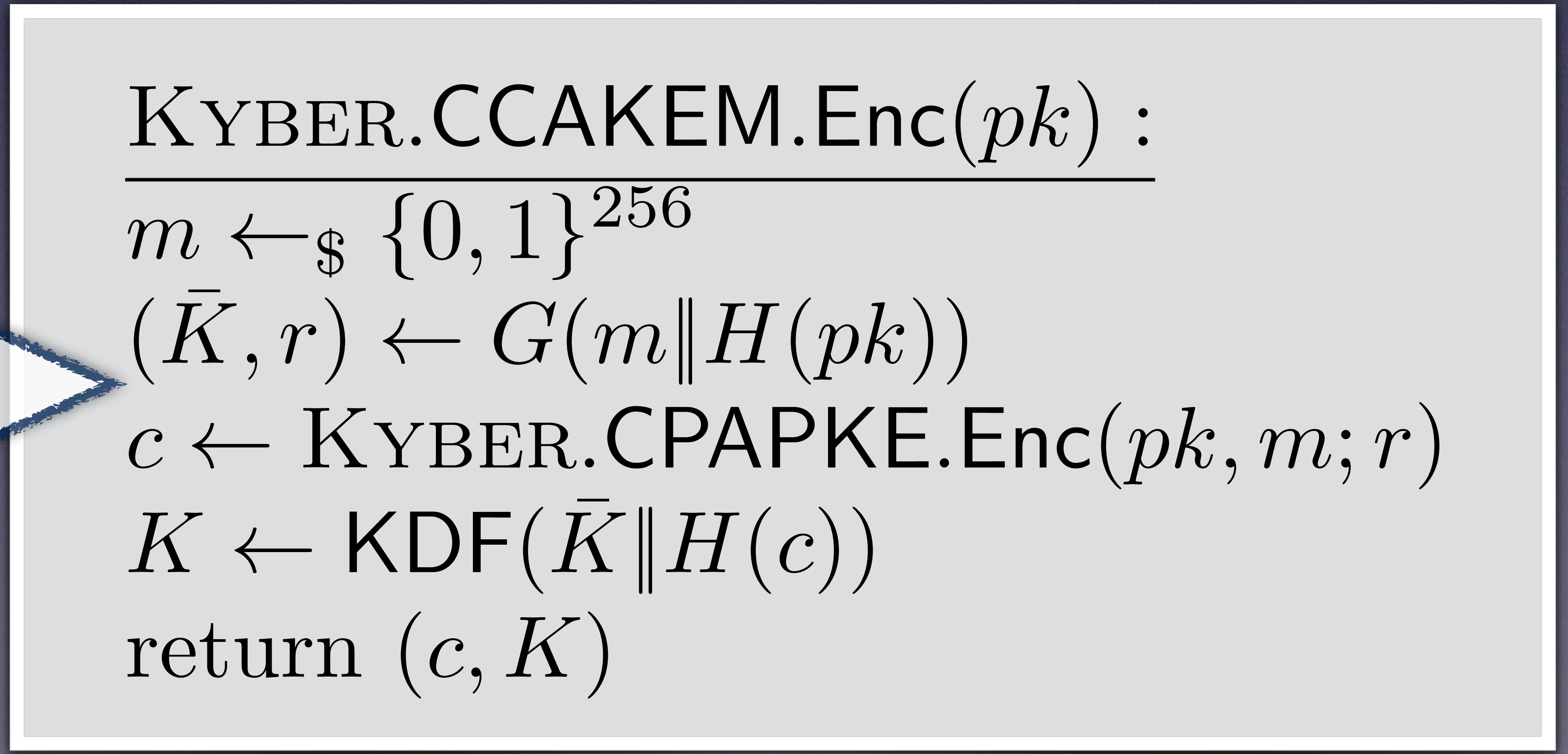
omitted ciphertext compression/decompression

$m = Dec(v - \mathbf{s}^T\mathbf{u})$



Kyber.CCAKEM: CCA-secure KEM via tweaked FO transform

- Use implicit rejection
- Hash public key into seed and shared key
- Hash ciphertext into shared key
- Use Keccak-based functions for all hashes and XOF



```
KYBER.CCAKEM.Enc($pk$) :
$m \leftarrow_{\$} \{0,1\}^{256}$
$(\bar{K}, r) \leftarrow G(m \| H(pk))$
$c \leftarrow$ KYBER.CPAPKE.Enc($pk, m; r$)
$K \leftarrow \mathrm{KDF}(\bar{K} \| H(c))$
return $(c, K)$
```

https://pq-crystals.org/kyber/data/slides-nistpqc19-schwabe.pdf

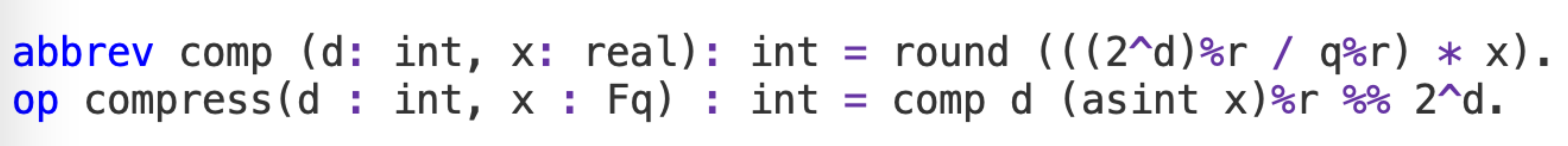
Specification goals

- Humans need to be able to check
    - Syntactically as close as possible to paper specification
- Prove properties of various operations stated in paper specification:
    - NTT description is correct and commutes with ring multiplication
    - Compression and decompression have claimed properties
    - Sampling procedures generate claimed distributions

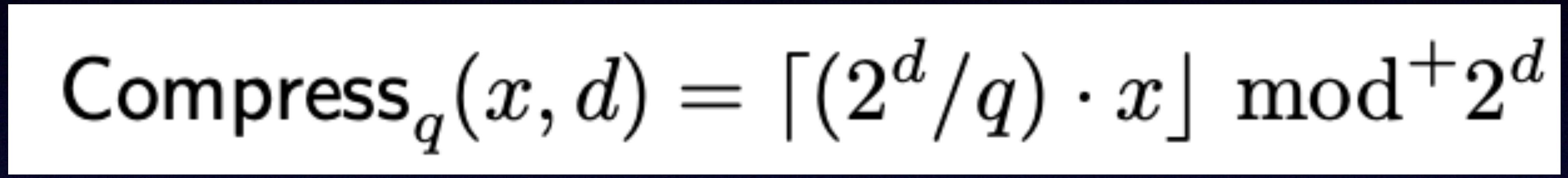
Specification non goals

- Executable spec:
  - generate test vectors
  - check the spec itself (?)
- Two solutions
  - Prove spec equivalent to HACSpec executable spec (ongoing)
  - Add an execution engine to EasyCrypt (future work)

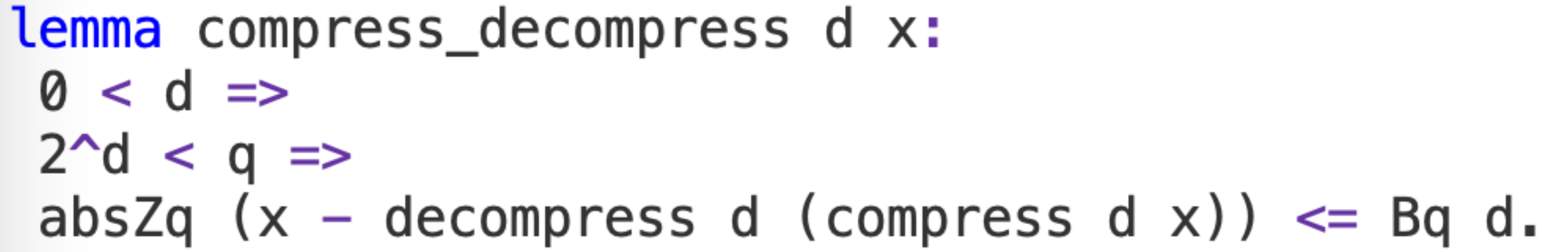
Examples



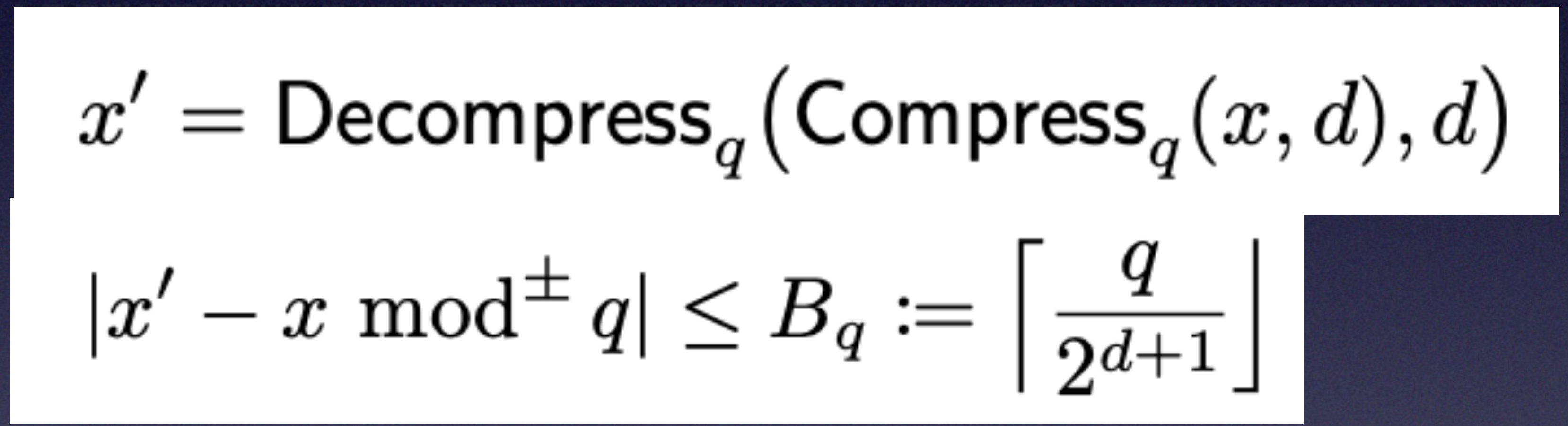
```
abbrev comp (d: int, x: real): int = round (((2^d)%r / q%r) * x).
op compress(d : int, x : Fq) : int = comp d (asint x)%r %% 2^d.
```



$$\mathsf{Compress}_q(x,d) = \lceil (2^d/q)\cdot x \rfloor \bmod^{+} 2^d$$

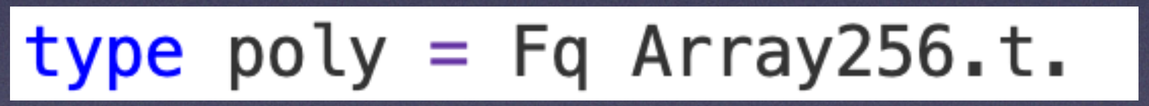


```
lemma compress_decompress d x:
 0 < d =>
 2^d < q =>
 absZq (x - decompress d (compress d x)) <= Bq d.
```

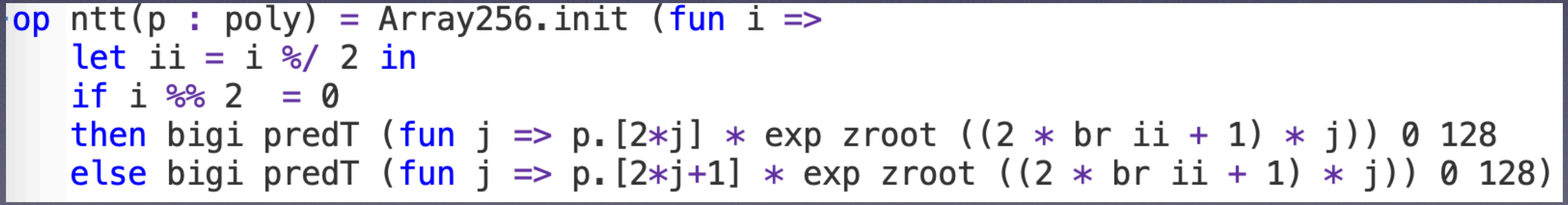


$$x' = \mathsf{Decompress}_q(\mathsf{Compress}_q(x,d),d)$$

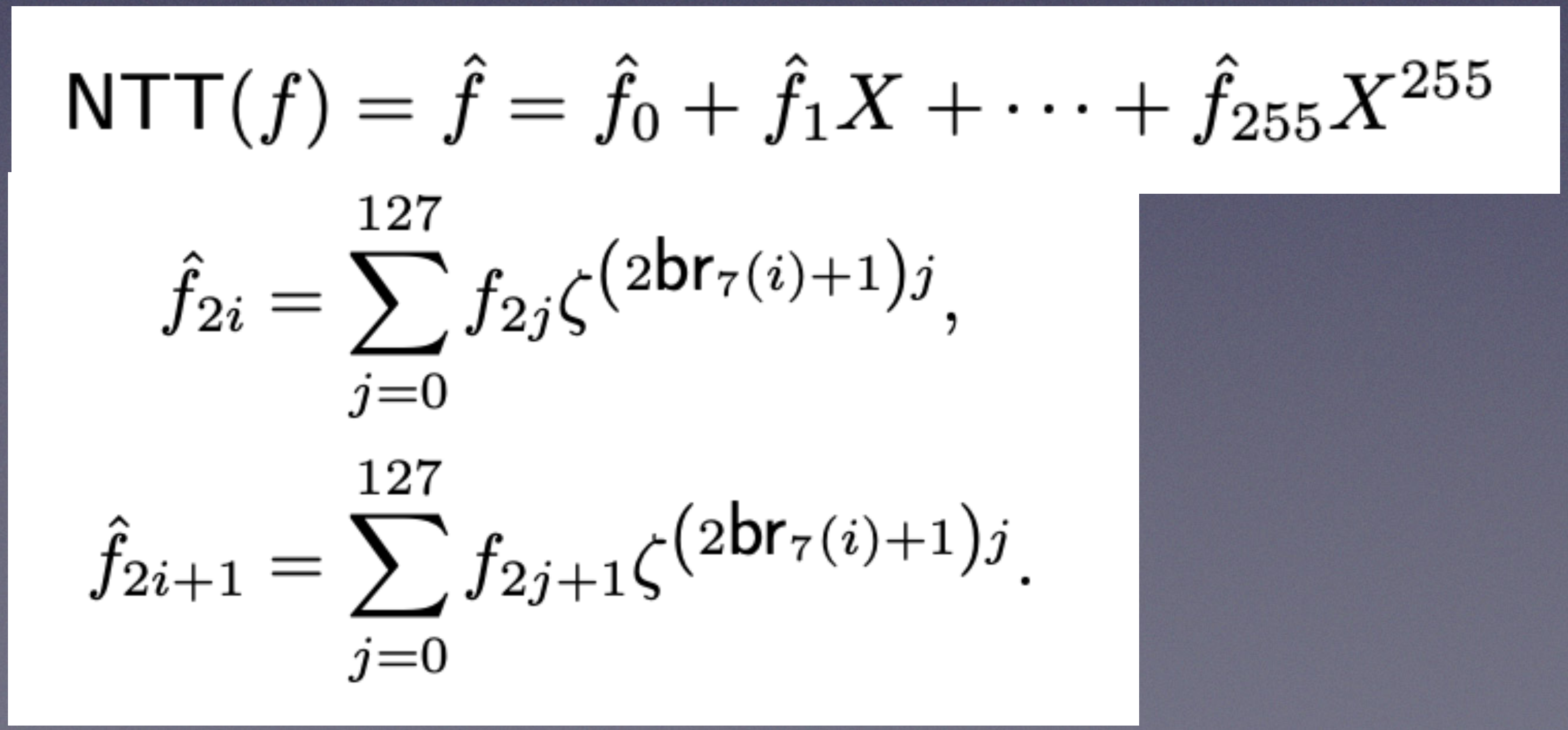
$$|x' - x \bmod^{\pm} q| \le B_q := \left\lceil \frac{q}{2^{d+1}} \right\rfloor$$



```
type poly = Fq Array256.t.
```



```
op ntt(p : poly) = Array256.init (fun i =>
   let ii = i %/ 2 in
   if i %% 2  = 0
   then bigi predT (fun j => p.[2*j] * exp zroot ((2 * br ii + 1) * j)) 0 128
   else bigi predT (fun j => p.[2*j+1] * exp zroot ((2 * br ii + 1) * j)) 0 128).
```



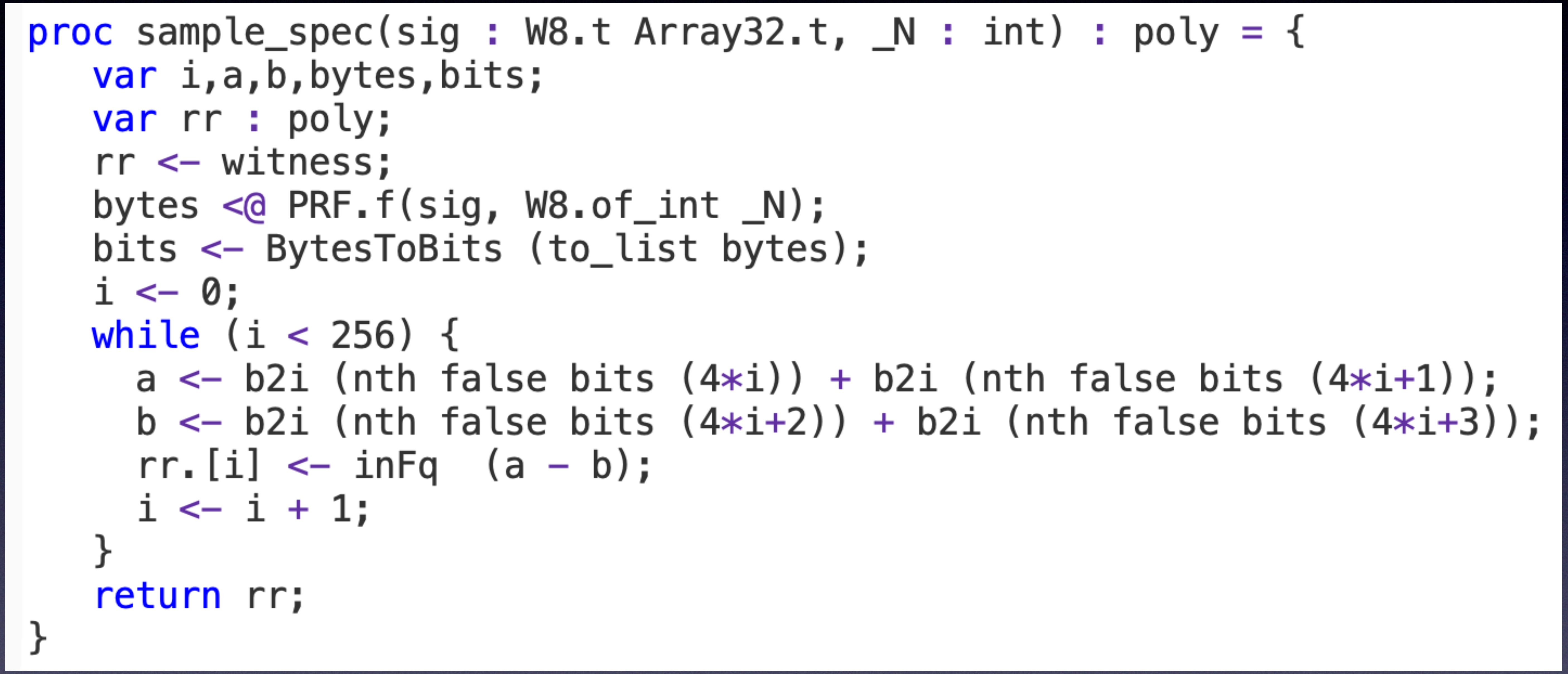
$$\mathsf{NTT}(f) = \hat{f} = \hat{f}_0 + \hat{f}_1 X + \cdots + \hat{f}_{255} X^{255}$$

$$\hat{f}_{2i} = \sum_{j=0}^{127} f_{2j}\zeta^{(2\mathsf{br}_7(i)+1)j},$$

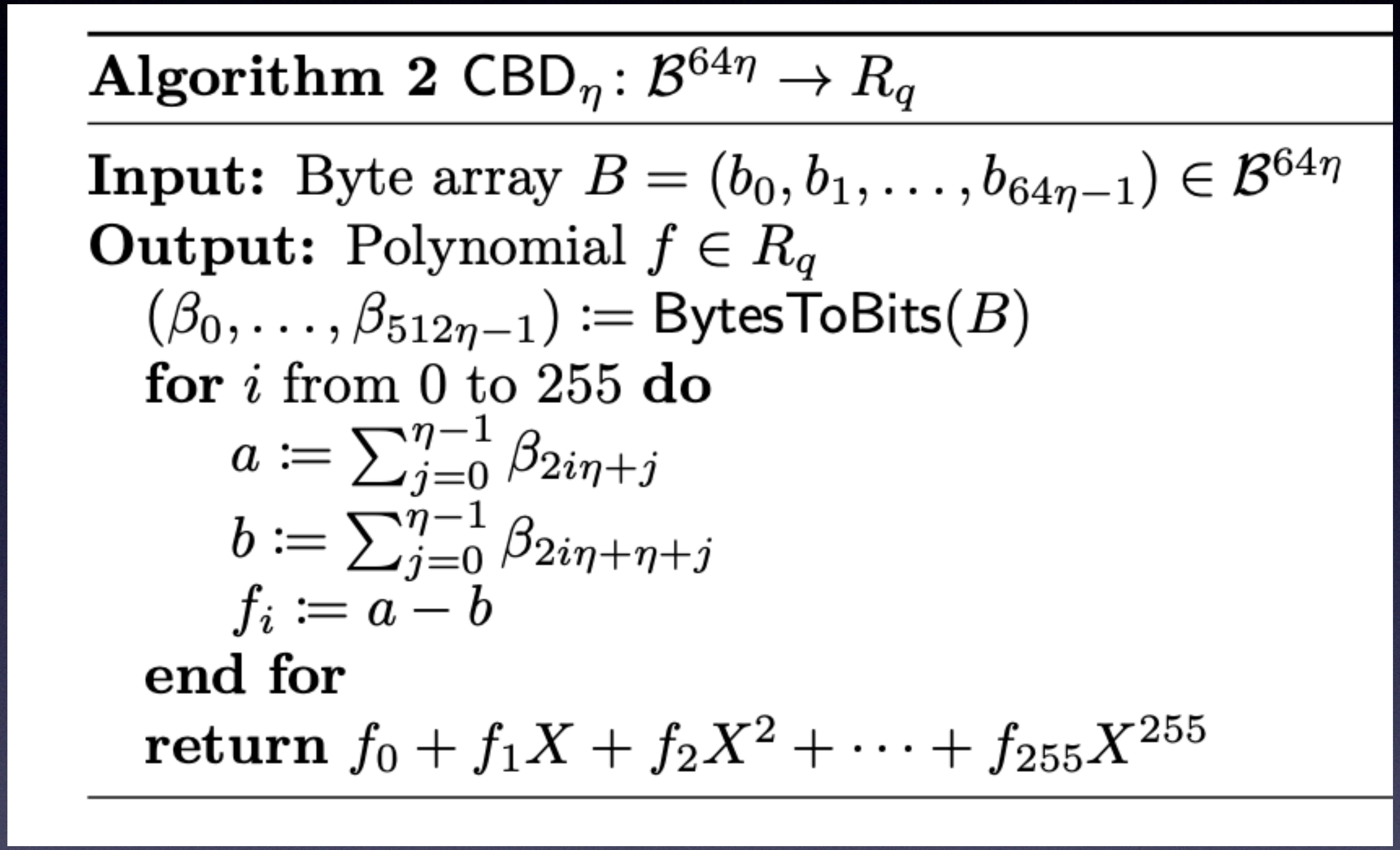
$$\hat{f}_{2i+1} = \sum_{j=0}^{127} f_{2j+1}\zeta^{(2\mathsf{br}_7(i)+1)j}.$$

```
lemma invnttK : cancel ntt invntt.
```

Examples



```
proc sample_spec(sig : W8.t Array32.t, _N : int) : poly = {
   var i,a,b,bytes,bits;
   var rr : poly;
   rr <- witness;
   bytes <@ PRF.f(sig, W8.of_int _N);
   bits <- BytesToBits (to_list bytes);
   i <- 0;
   while (i < 256) {
     a <- b2i (nth false bits (4*i)) + b2i (nth false bits (4*i+1));
     b <- b2i (nth false bits (4*i+2)) + b2i (nth false bits (4*i+3));
     rr.[i] <- inFq  (a - b);
     i <- i + 1;
   }
   return rr;
}
```

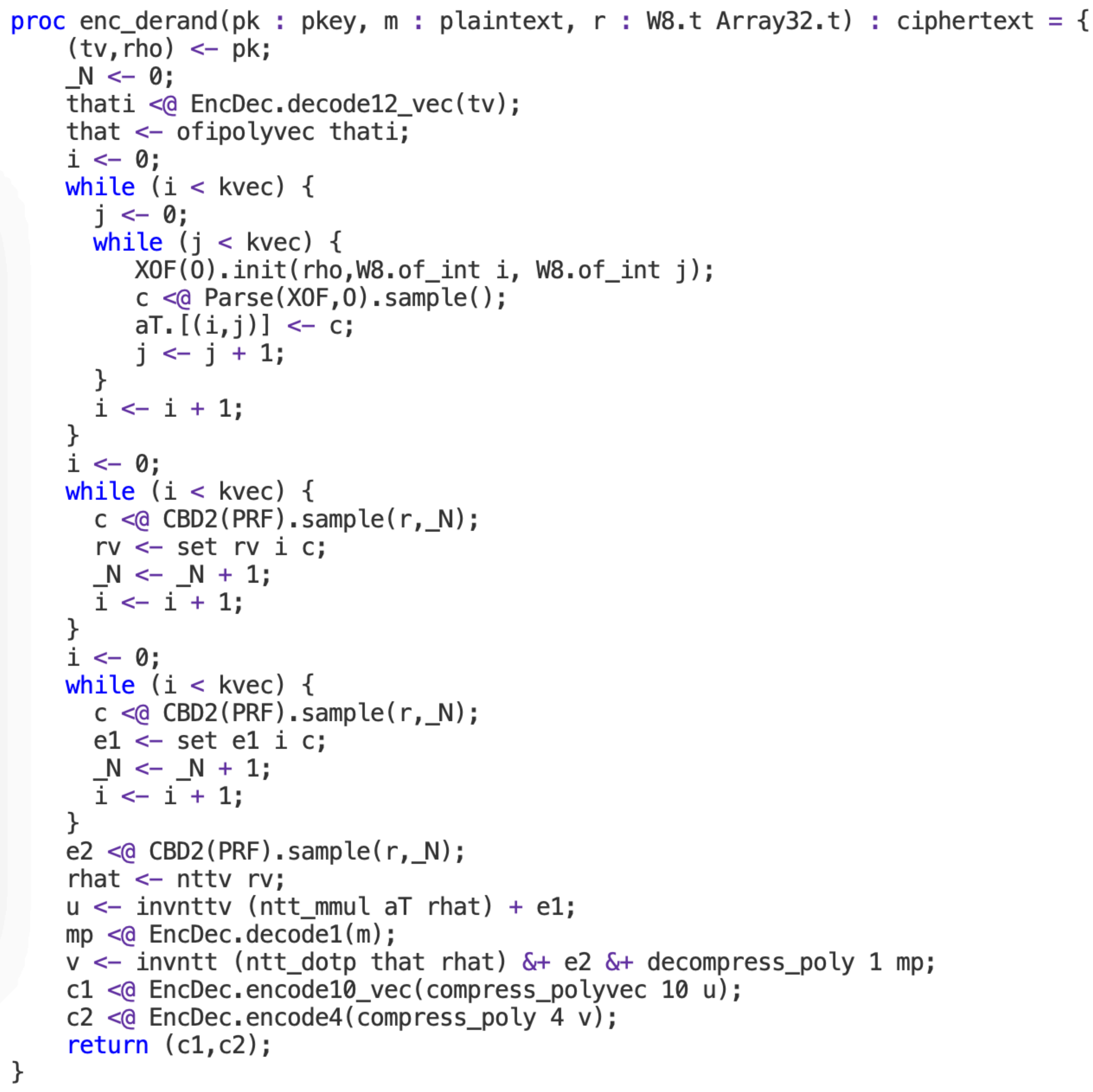


```
Algorithm 2 CBD$_\eta$: $\mathcal{B}^{64\eta} \rightarrow R_q$
Input: Byte array $B = (b_0, b_1, \ldots, b_{64\eta-1}) \in \mathcal{B}^{64\eta}$
Output: Polynomial $f \in R_q$
  $(\beta_0, \ldots, \beta_{512\eta-1}) := \mathsf{BytesToBits}(B)$
  for $i$ from 0 to 255 do
    $a := \sum_{j=0}^{\eta-1} \beta_{2i\eta+j}$
    $b := \sum_{j=0}^{\eta-1} \beta_{2i\eta+\eta+j}$
    $f_i := a - b$
  end for
  return $f_0 + f_1 X + f_2 X^2 + \cdots + f_{255} X^{255}$
```

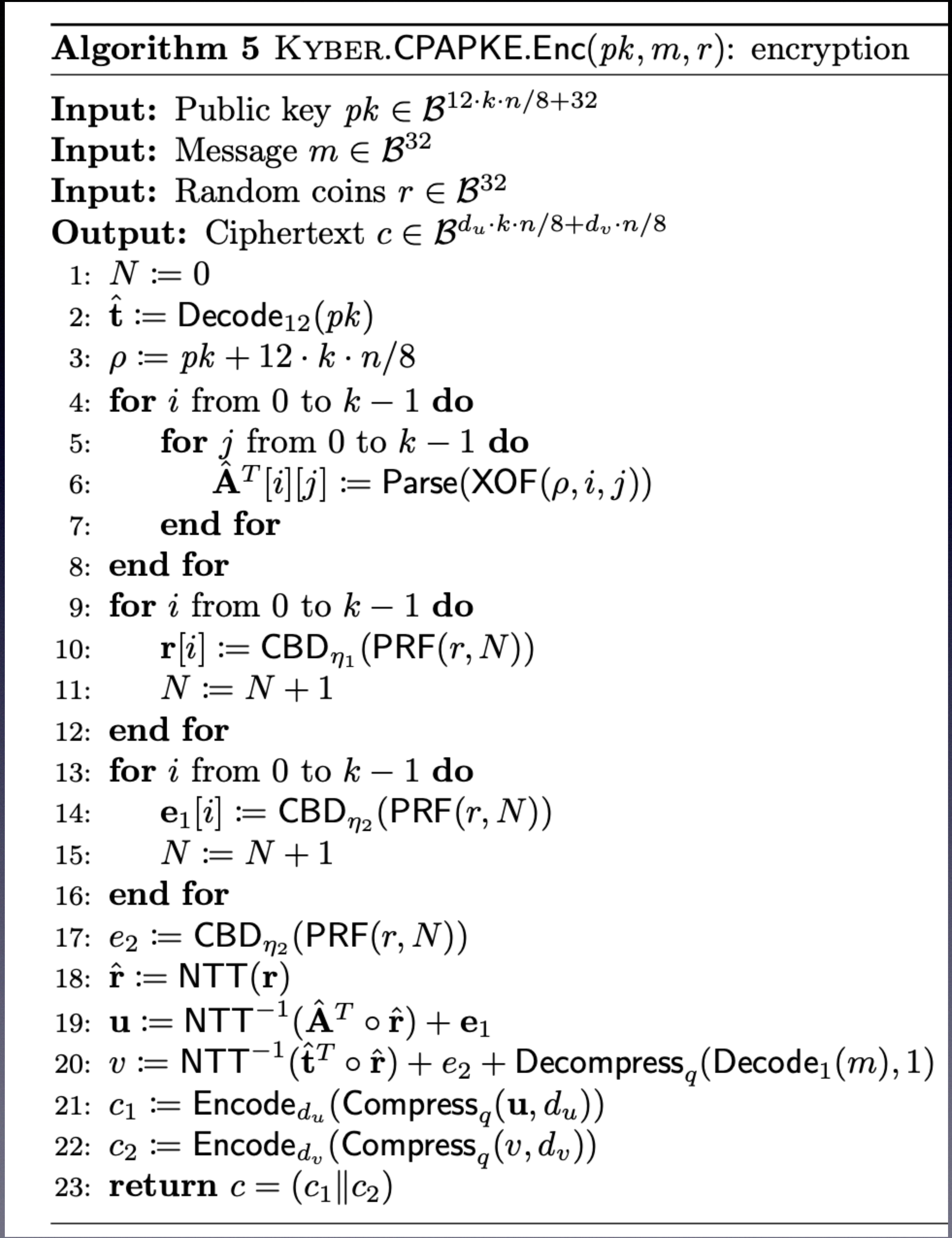
```
equiv CBD2rnd_equiv:
 CBD2rnd.sample_real ~ CBD2rnd.sample_ideal:
 true ==> ={res}.
```

Idealize PRF.f and prove procedure produces correct distribution over Rq: each coeff. independently sampled from binomial distribution.

Examples



```
proc enc_derand(pk : pkey, m : plaintext, r : W8.t Array32.t) : ciphertext = {
    (tv,rho) <- pk;
    _N <- 0;
    thati <@ EncDec.decode12_vec(tv);
    that <- ofipolyvec thati;
    i <- 0;
    while (i < kvec) {
      j <- 0;
      while (j < kvec) {
         XOF(O).init(rho,W8.of_int i, W8.of_int j);
         c <@ Parse(XOF,O).sample();
         aT.[(i,j)] <- c;
         j <- j + 1;
      }
      i <- i + 1;
    }
    i <- 0;
    while (i < kvec) {
      c <@ CBD2(PRF).sample(r,_N);
      rv <- set rv i c;
      _N <- _N + 1;
      i <- i + 1;
    }
    i <- 0;
    while (i < kvec) {
      c <@ CBD2(PRF).sample(r,_N);
      e1 <- set e1 i c;
      _N <- _N + 1;
      i <- i + 1;
    }
    e2 <@ CBD2(PRF).sample(r,_N);
    rhat <- nttv rv;
    u <- invnttv (ntt_mmul aT rhat) + e1;
    mp <@ EncDec.decode1(m);
    v <- invntt (ntt_dotp that rhat) &+ e2 &+ decompress_poly 1 mp;
    c1 <@ EncDec.encode10_vec(compress_polyvec 10 u);
    c2 <@ EncDec.encode4(compress_poly 4 v);
    return (c1,c2);
}
```



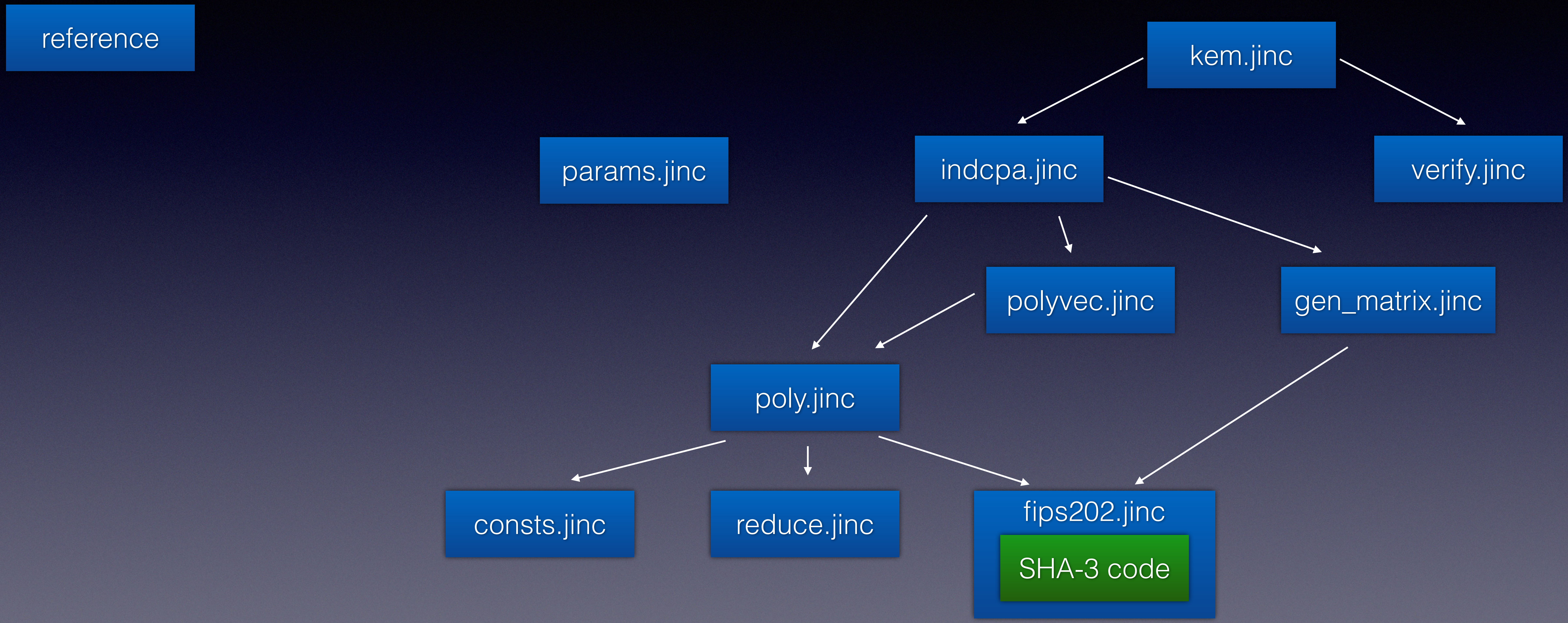
Algorithm 5 KYBER.CPAPKE.Enc(pk, m, r): encryption

```
Input: Public key $pk \in \mathcal{B}^{12 \cdot k \cdot n/8+32}$
Input: Message $m \in \mathcal{B}^{32}$
Input: Random coins $r \in \mathcal{B}^{32}$
Output: Ciphertext $c \in \mathcal{B}^{d_u \cdot k \cdot n/8 + d_v \cdot n/8}$
$N := 0$
$\hat{\mathbf{t}} := \mathsf{Decode}_{12}(pk)$
$\rho := pk + 12 \cdot k \cdot n/8$
for $i$ from 0 to $k-1$ do
    for $j$ from 0 to $k-1$ do
        $\hat{\mathbf{A}}^T[i][j] := \mathsf{Parse}(\mathsf{XOF}(\rho, i, j))$
    end for
end for
for $i$ from 0 to $k-1$ do
    $\mathbf{r}[i] := \mathsf{CBD}_{\eta_1}(\mathsf{PRF}(r, N))$
    $N := N + 1$
end for
for $i$ from 0 to $k-1$ do
    $\mathbf{e}_1[i] := \mathsf{CBD}_{\eta_2}(\mathsf{PRF}(r, N))$
    $N := N + 1$
end for
$e_2 := \mathsf{CBD}_{\eta_2}(\mathsf{PRF}(r, N))$
$\hat{\mathbf{r}} := \mathsf{NTT}(\mathbf{r})$
$\mathbf{u} := \mathsf{NTT}^{-1}(\hat{\mathbf{A}}^T \circ \hat{\mathbf{r}}) + \mathbf{e}_1$
$v := \mathsf{NTT}^{-1}(\hat{\mathbf{t}}^T \circ \hat{\mathbf{r}}) + e_2 + \mathsf{Decompress}_q(\mathsf{Decode}_1(m), 1)$
$c_1 := \mathsf{Encode}_{d_u}(\mathsf{Compress}_q(\mathbf{u}, d_u))$
$c_2 := \mathsf{Encode}_{d_v}(\mathsf{Compress}_q(v, d_v))$
return $c = (c_1 \| c_2)$
```

Jasmin Implementation

- Structure of the code
- Performance
- Snippets/examples

Structure of Jasmin code



reference

kem.jinc

params.jinc

indcpa.jinc

verify.jinc

polyvec.jinc

gen_matrix.jinc

poly.jinc

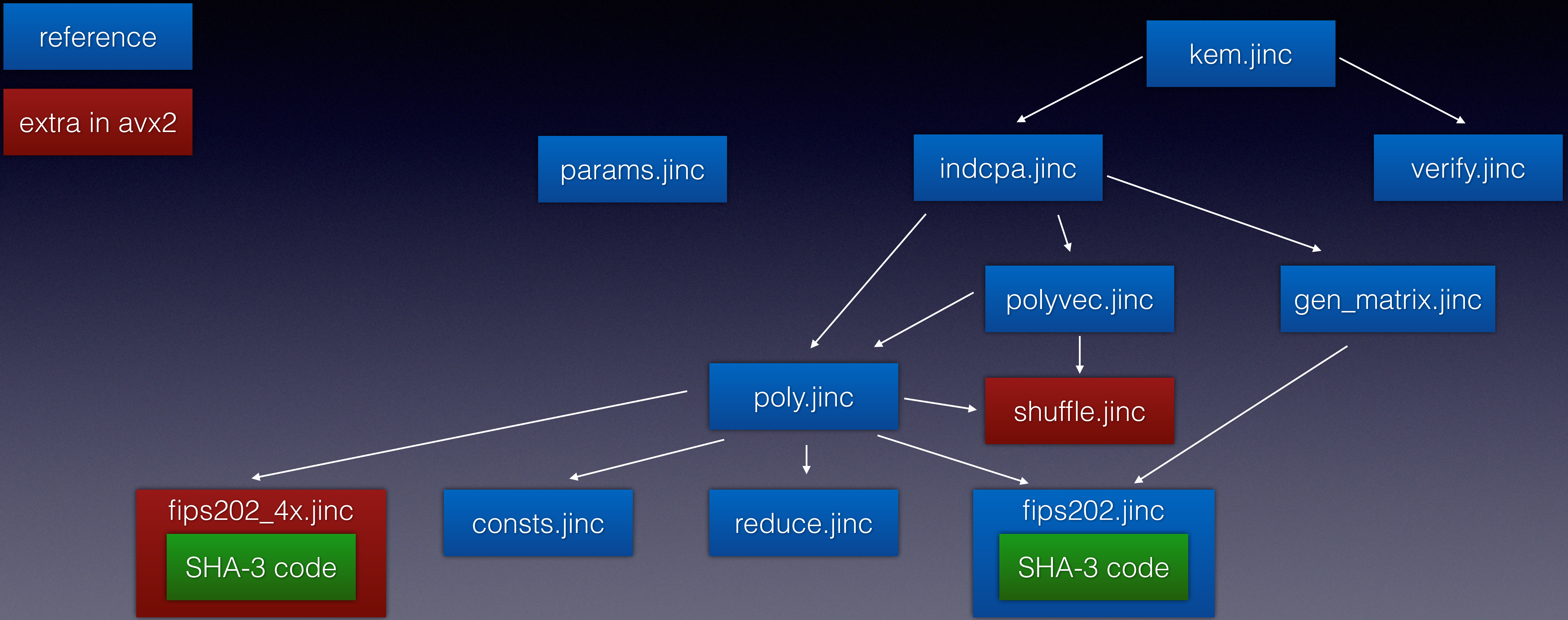
consts.jinc

reduce.jinc

fips202.jinc

SHA-3 code

Structure of Jasmin code



reference

extra in avx2

kem.jinc

params.jinc

indcpa.jinc

verify.jinc

polyvec.jinc

gen_matrix.jinc

poly.jinc

shuffle.jinc

fips202_4x.jinc

SHA-3 code

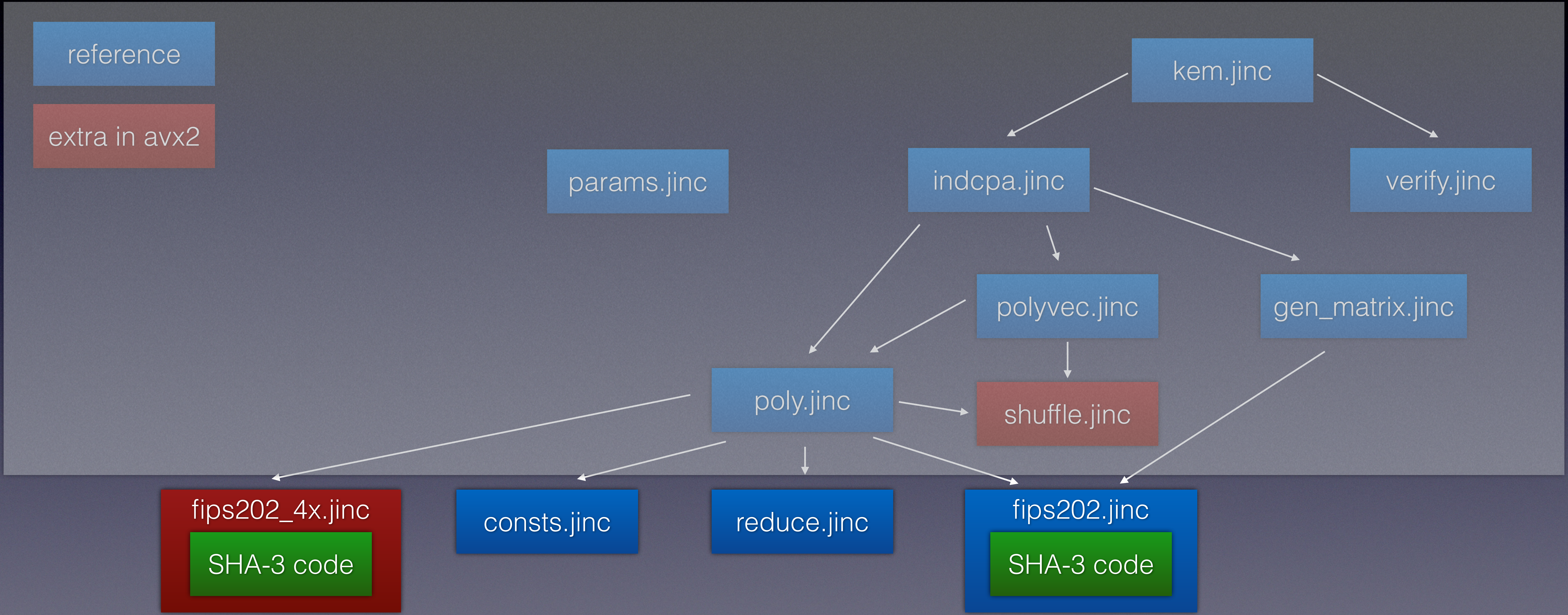
consts.jinc

reduce.jinc

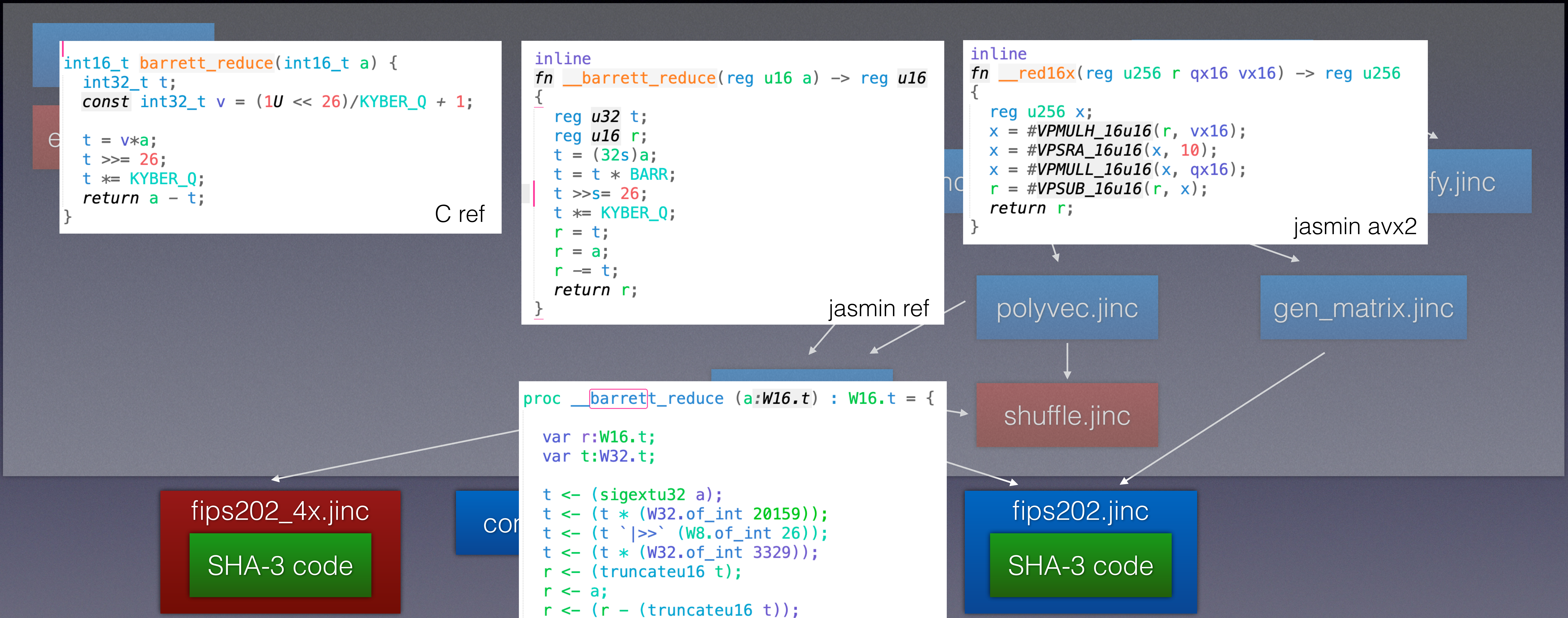
fips202.jinc

SHA-3 code

Structure of Jasmin code



Structure of Jasmin code



```c
int16_t barrett_reduce(int16_t a) {
  int32_t t;
  const int32_t v = (1U << 26)/KYBER_Q + 1;

  t = v*a;
  t >>= 26;
  t *= KYBER_Q;
  return a - t;
}
```

C ref

```
inline
fn __barrett_reduce(reg u16 a) -> reg u16
{
  reg u32 t;
  reg u16 r;
  t = (32s)a;
  t = t * BARR;
  t >>s= 26;
  t *= KYBER_Q;
  r = t;
  r = a;
  r -= t;
  return r;
}
```

jasmin ref

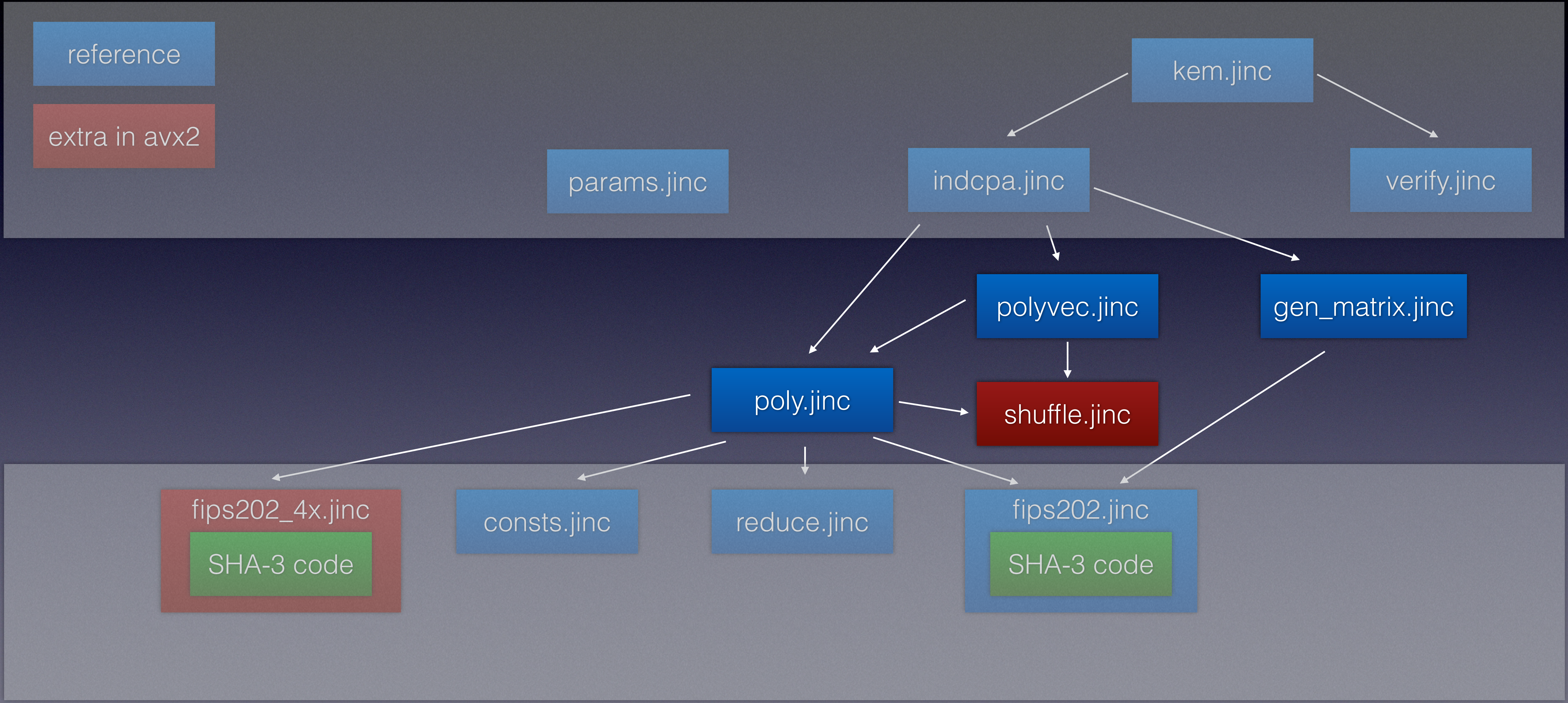
```
inline
fn __red16x(reg u256 r qx16 vx16) -> reg u256
{
  reg u256 x;
  x = #VPMULH_16u16(r, vx16);
  x = #VPSRA_16u16(x, 10);
  x = #VPMULL_16u16(x, qx16);
  r = #VPSUB_16u16(r, x);
  return r;
}
```

jasmin avx2

```
proc __barrett_reduce (a:W16.t) : W16.t = {

  var r:W16.t;
  var t:W32.t;

  t <- (sigextu32 a);
  t <- (t * (W32.of_int 20159));
  t <- (t `|>>` (W8.of_int 26));
  t <- (t * (W32.of_int 3329));
  r <- (truncateu16 t);
  r <- a;
  r <- (r - (truncateu16 t));
  return (r);
}
```

Easycrypt ref

Structure of Jasmin code



reference

extra in avx2

kem.jinc

params.jinc

indcpa.jinc

verify.jinc

polyvec.jinc

gen_matrix.jinc

poly.jinc

shuffle.jinc

fips202_4x.jinc

SHA-3 code

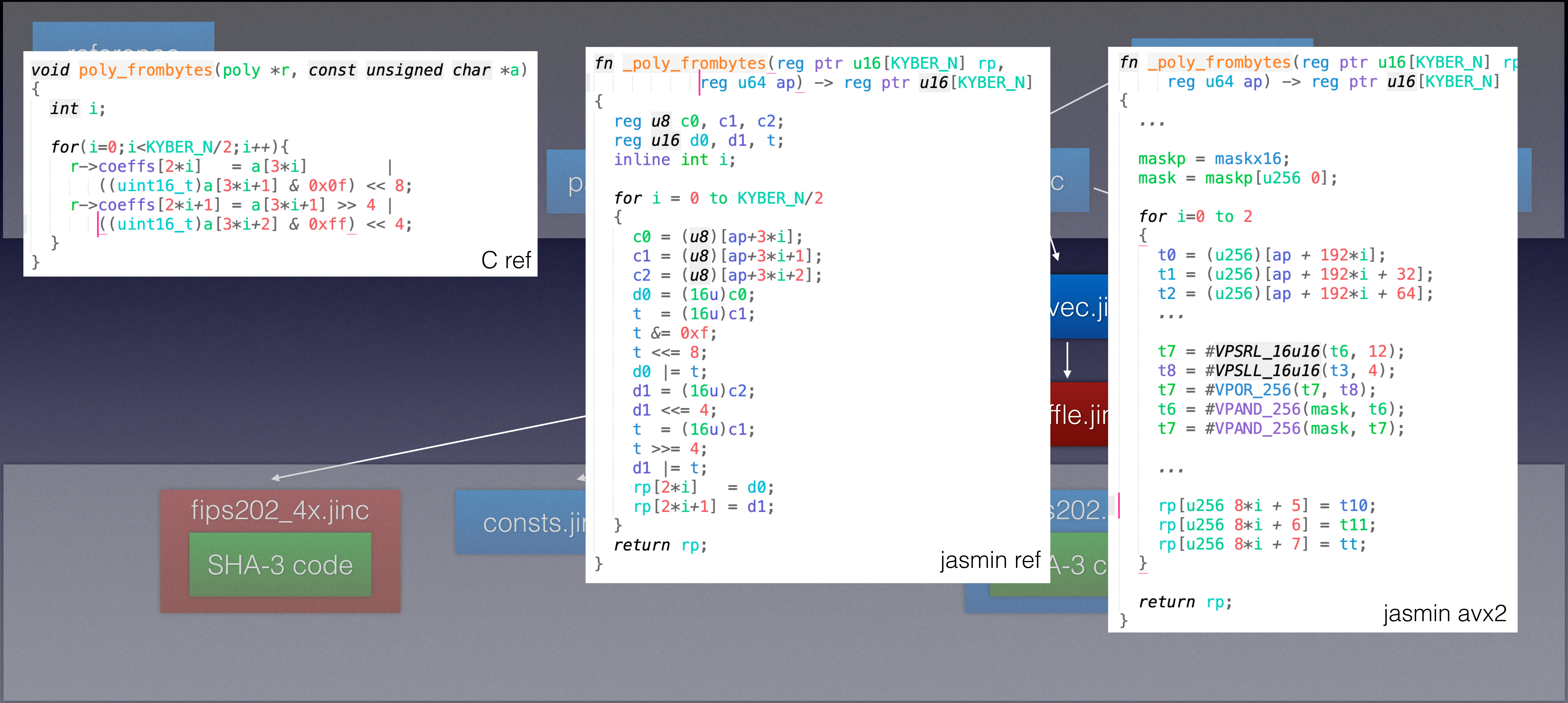
consts.jinc

reduce.jinc

fips202.jinc

SHA-3 code

Structure of Jasmin code



```
void poly_frombytes(poly *r, const unsigned char *a)
{
  int i;

  for(i=0;i<KYBER_N/2;i++){
    r->coeffs[2*i]   = a[3*i]         |
       ((uint16_t)a[3*i+1] & 0x0f) << 8;
    r->coeffs[2*i+1] = a[3*i+1] >> 4 |
       ((uint16_t)a[3*i+2] & 0xff) << 4;
  }
}
```

C ref

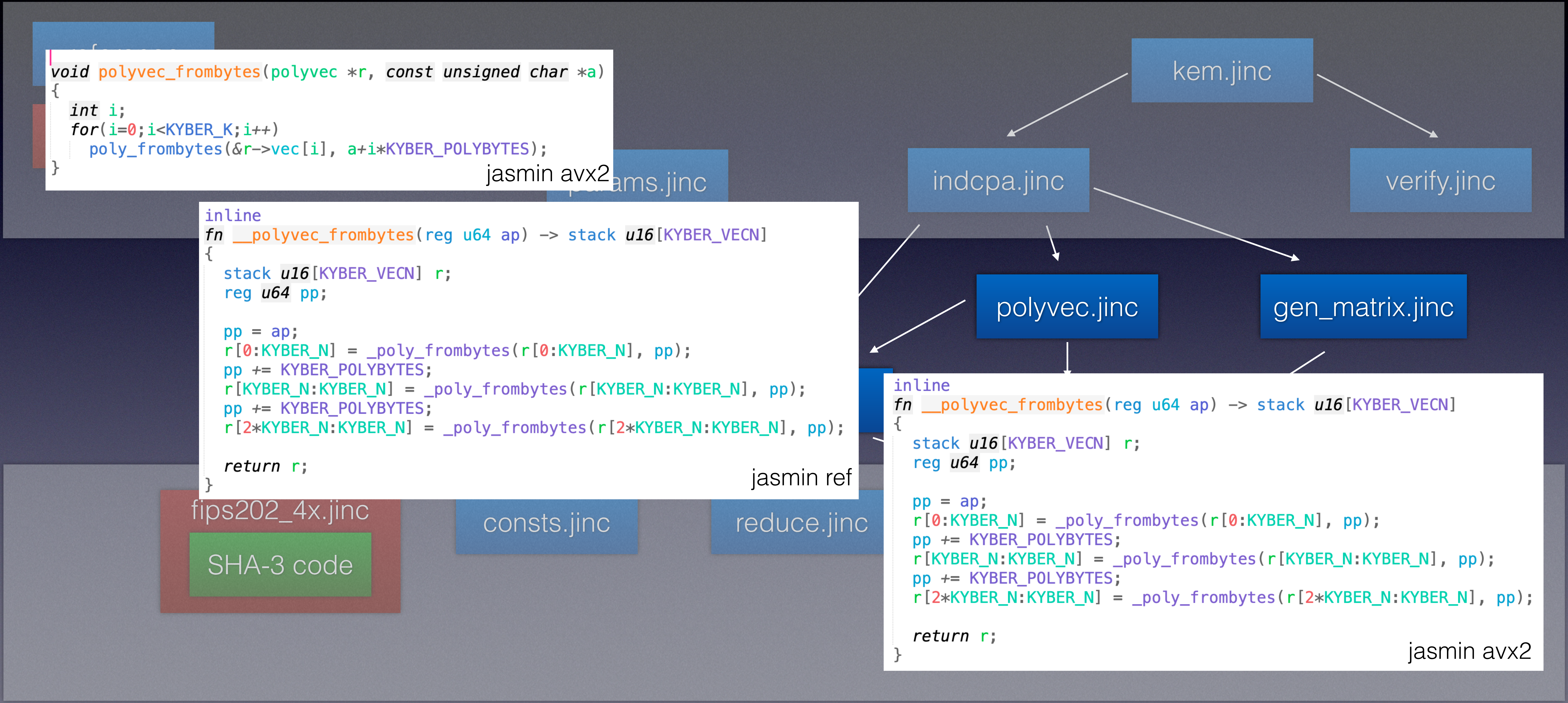
```
fn _poly_frombytes(reg ptr u16[KYBER_N] rp,
                   reg u64 ap) -> reg ptr u16[KYBER_N]
{
  reg u8 c0, c1, c2;
  reg u16 d0, d1, t;
  inline int i;

  for i = 0 to KYBER_N/2
  {
    c0 = (u8)[ap+3*i];
    c1 = (u8)[ap+3*i+1];
    c2 = (u8)[ap+3*i+2];
    d0 = (16u)c0;
    t  = (16u)c1;
    t &= 0xf;
    t <<= 8;
    d0 |= t;
    d1 = (16u)c2;
    d1 <<= 4;
    t  = (16u)c1;
    t >>= 4;
    d1 |= t;
    rp[2*i]   = d0;
    rp[2*i+1] = d1;
  }
  return rp;
}
```

jasmin ref

```
fn _poly_frombytes(reg ptr u16[KYBER_N] r
   reg u64 ap) -> reg ptr u16[KYBER_N]
{
  ...

  maskp = maskx16;
  mask = maskp[u256 0];

  for i=0 to 2
  {
    t0 = (u256)[ap + 192*i];
    t1 = (u256)[ap + 192*i + 32];
    t2 = (u256)[ap + 192*i + 64];
    ...

    t7 = #VPSRL_16u16(t6, 12);
    t8 = #VPSLL_16u16(t3, 4);
    t7 = #VPOR_256(t7, t8);
    t6 = #VPAND_256(mask, t6);
    t7 = #VPAND_256(mask, t7);

    ...

    rp[u256 8*i + 5] = t10;
    rp[u256 8*i + 6] = t11;
    rp[u256 8*i + 7] = tt;
  }

  return rp;
}
```

jasmin avx2

Structure of Jasmin code



```
void polyvec_frombytes(polyvec *r, const unsigned char *a)
{
  int i;
  for(i=0;i<KYBER_K;i++)
    poly_frombytes(&r->vec[i], a+i*KYBER_POLYBYTES);
}
```

jasmin avx2

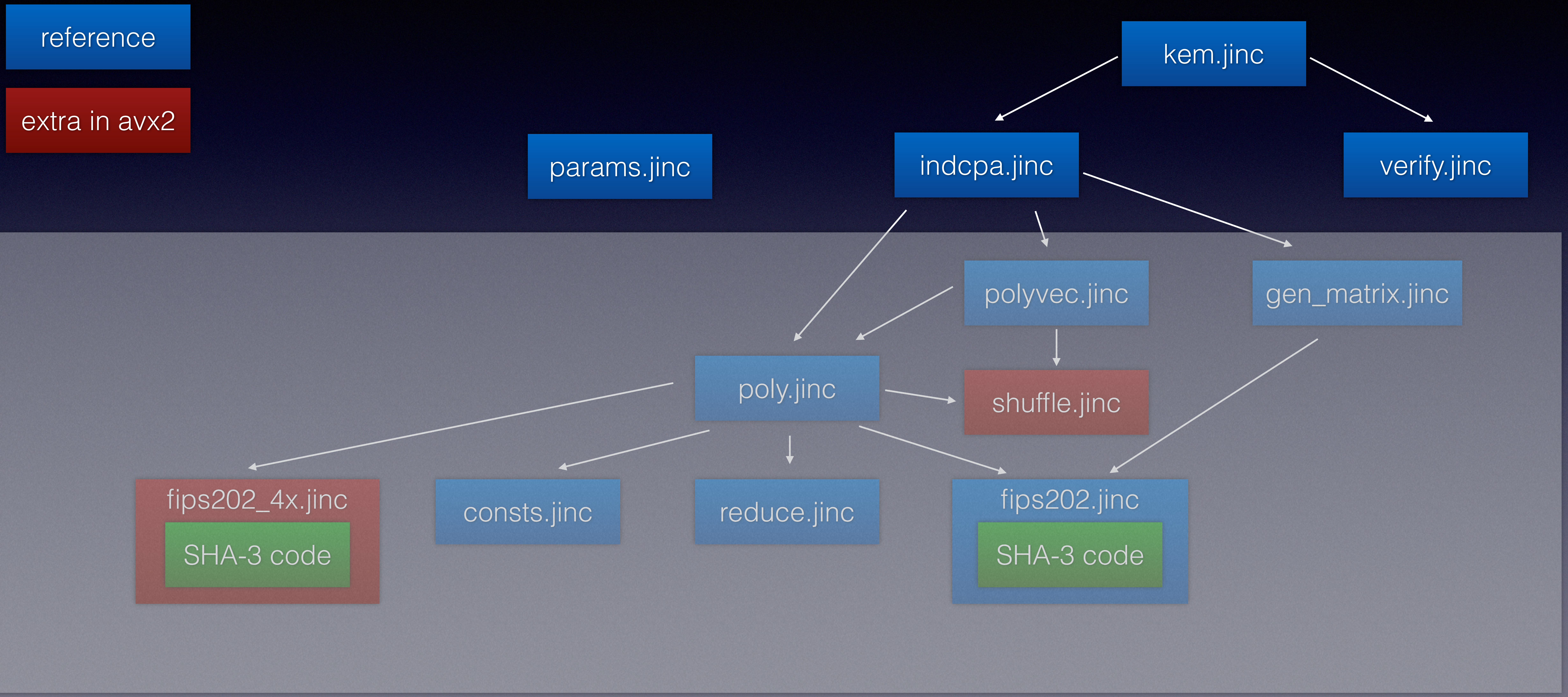
```
inline
fn __polyvec_frombytes(reg u64 ap) -> stack u16[KYBER_VECN]
{
  stack u16[KYBER_VECN] r;
  reg u64 pp;

  pp = ap;
  r[0:KYBER_N] = _poly_frombytes(r[0:KYBER_N], pp);
  pp += KYBER_POLYBYTES;
  r[KYBER_N:KYBER_N] = _poly_frombytes(r[KYBER_N:KYBER_N], pp);
  pp += KYBER_POLYBYTES;
  r[2*KYBER_N:KYBER_N] = _poly_frombytes(r[2*KYBER_N:KYBER_N], pp);

  return r;
}
```

jasmin ref

```
inline
fn __polyvec_frombytes(reg u64 ap) -> stack u16[KYBER_VECN]
{
  stack u16[KYBER_VECN] r;
  reg u64 pp;

  pp = ap;
  r[0:KYBER_N] = _poly_frombytes(r[0:KYBER_N], pp);
  pp += KYBER_POLYBYTES;
  r[KYBER_N:KYBER_N] = _poly_frombytes(r[KYBER_N:KYBER_N], pp);
  pp += KYBER_POLYBYTES;
  r[2*KYBER_N:KYBER_N] = _poly_frombytes(r[2*KYBER_N:KYBER_N], pp);

  return r;
}
```

jasmin avx2

Structure of Jasmin code



reference

extra in avx2

kem.jinc

params.jinc

indcpa.jinc

verify.jinc

polyvec.jinc

gen_matrix.jinc

poly.jinc

shuffle.jinc

fips202_4x.jinc

SHA-3 code

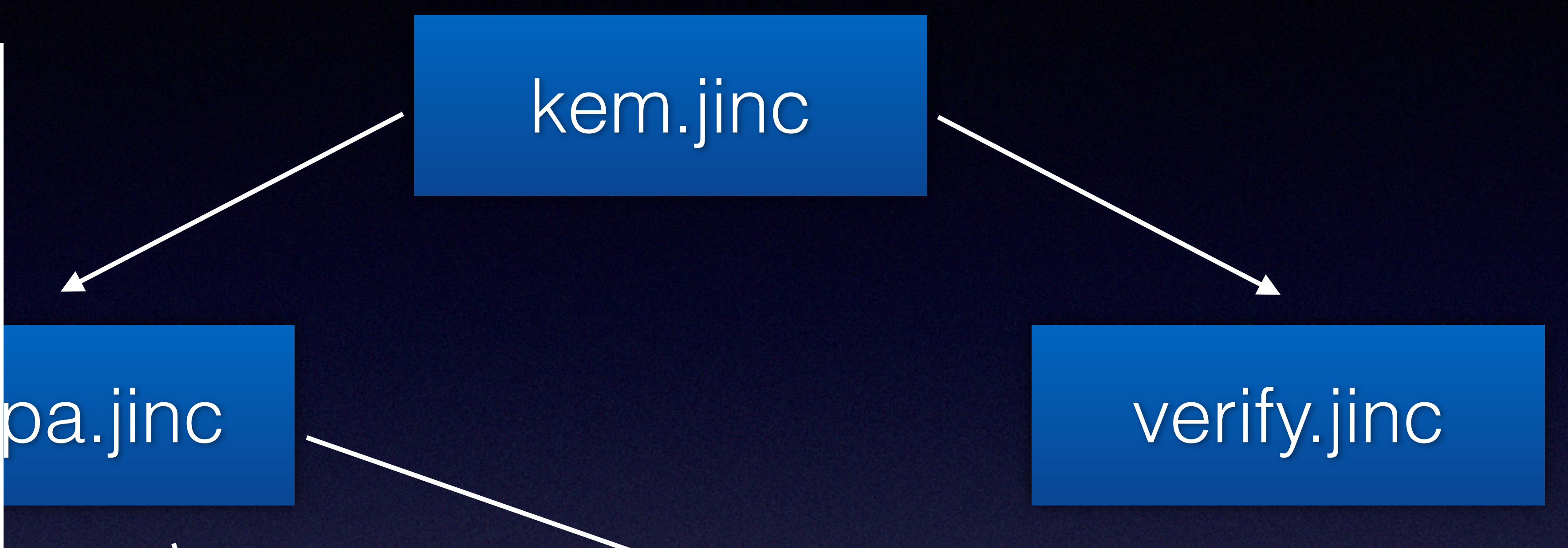
consts.jinc

reduce.jinc

fips202.jinc

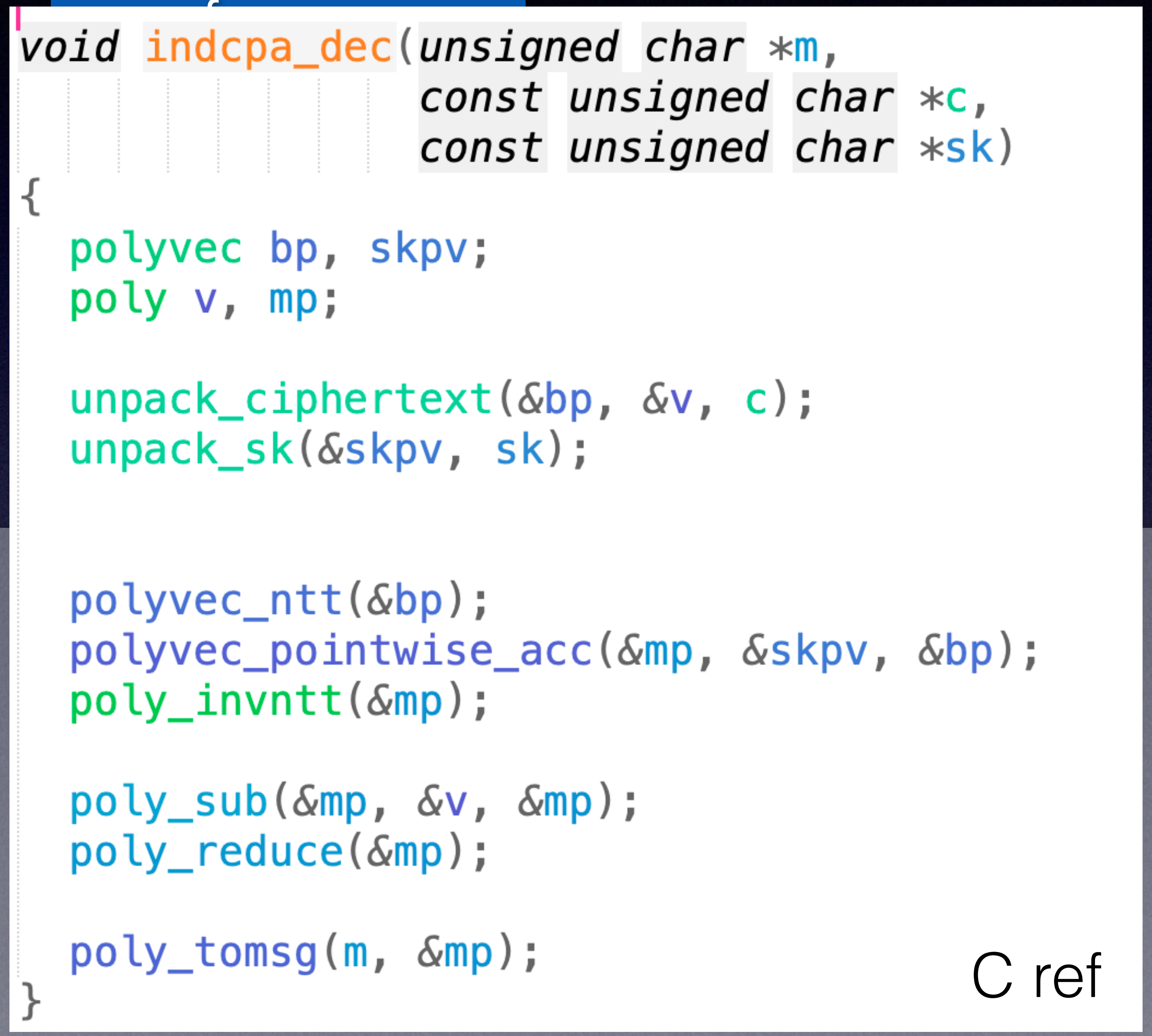
SHA-3 code

Structure of Jasmin code



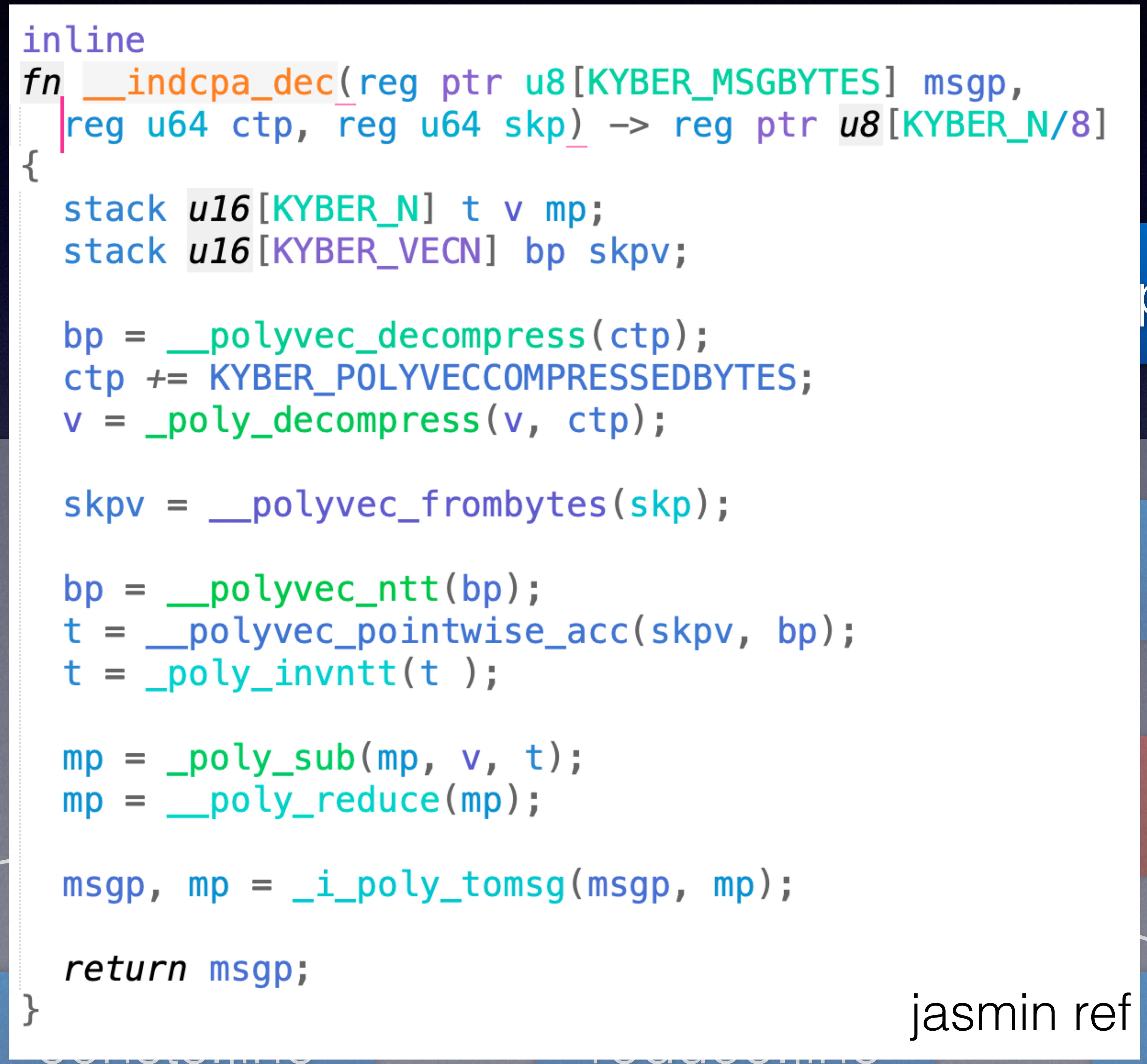
fips202_4x.jinc

SHA-3 code



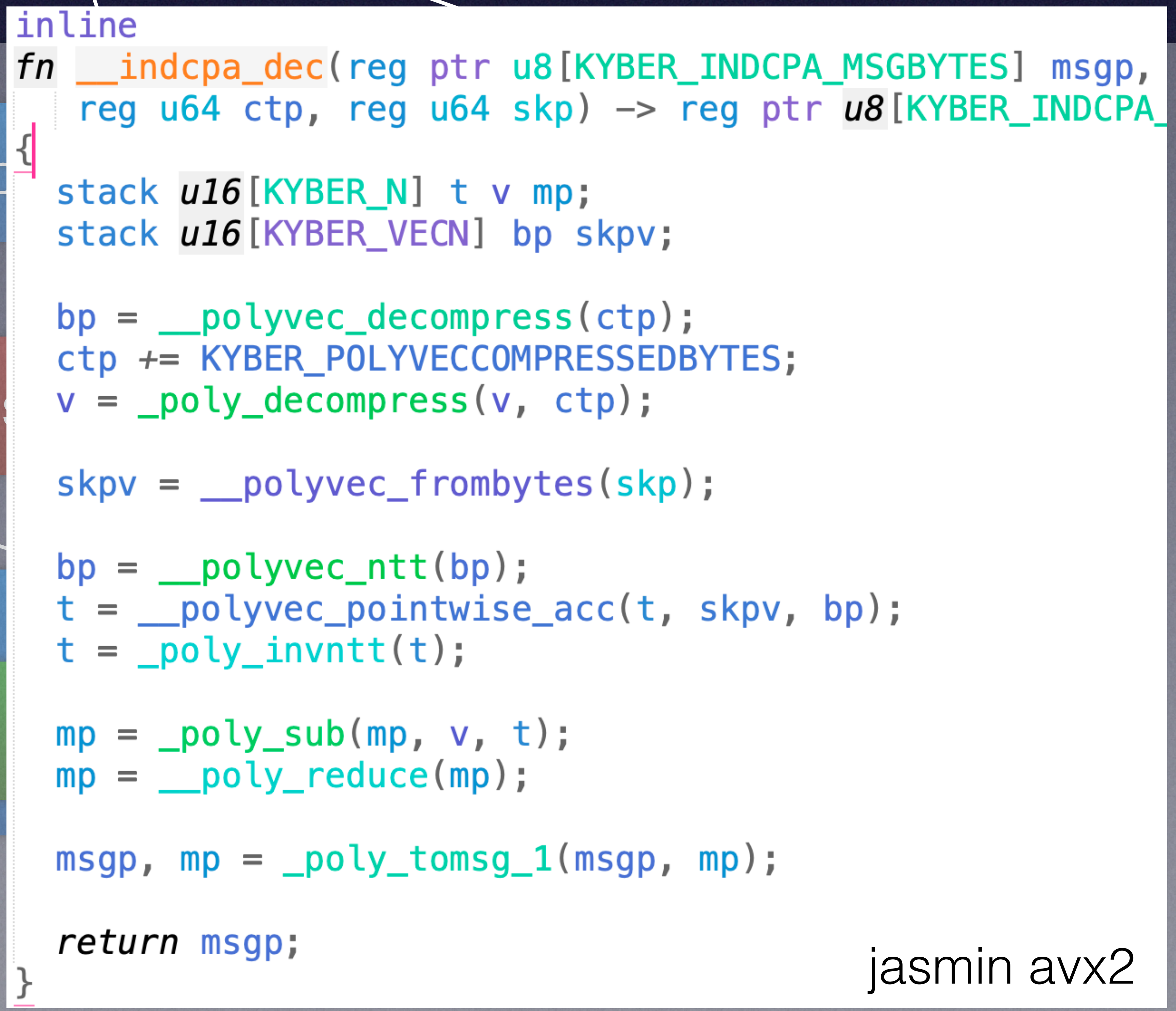
```
void indcpa_dec(unsigned char *m,
                const unsigned char *c,
                const unsigned char *sk)
{
  polyvec bp, skpv;
  poly v, mp;

  unpack_ciphertext(&bp, &v, c);
  unpack_sk(&skpv, sk);


  polyvec_ntt(&bp);
  polyvec_pointwise_acc(&mp, &skpv, &bp);
  poly_invntt(&mp);

  poly_sub(&mp, &v, &mp);
  poly_reduce(&mp);

  poly_tomsg(m, &mp);
}
```

C ref



```
inline
fn __indcpa_dec(reg ptr u8[KYBER_MSGBYTES] msgp,
  reg u64 ctp, reg u64 skp) -> reg ptr u8[KYBER_N/8]
{
  stack u16[KYBER_N] t v mp;
  stack u16[KYBER_VECN] bp skpv;

  bp = __polyvec_decompress(ctp);
  ctp += KYBER_POLYVECCOMPRESSEDBYTES;
  v = _poly_decompress(v, ctp);

  skpv = __polyvec_frombytes(skp);

  bp = __polyvec_ntt(bp);
  t = __polyvec_pointwise_acc(skpv, bp);
  t = _poly_invntt(t );

  mp = _poly_sub(mp, v, t);
  mp = __poly_reduce(mp);

  msgp, mp = _i_poly_tomsg(msgp, mp);

  return msgp;
}
```

jasmin ref

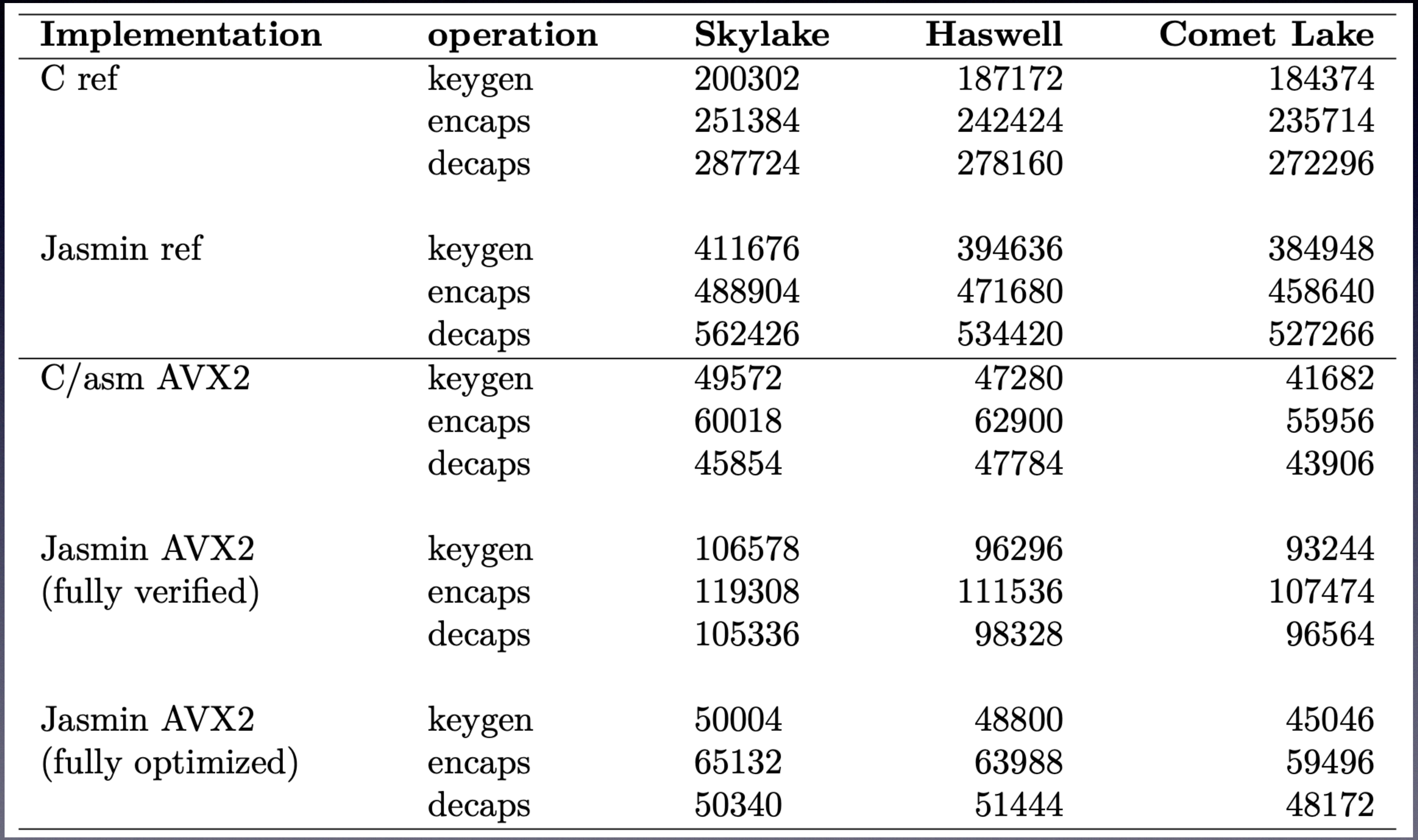


```
inline
fn __indcpa_dec(reg ptr u8[KYBER_INDCPA_MSGBYTES] msgp,
   reg u64 ctp, reg u64 skp) -> reg ptr u8[KYBER_INDCPA_
{
  stack u16[KYBER_N] t v mp;
  stack u16[KYBER_VECN] bp skpv;

  bp = __polyvec_decompress(ctp);
  ctp += KYBER_POLYVECCOMPRESSEDBYTES;
  v = _poly_decompress(v, ctp);

  skpv = __polyvec_frombytes(skp);

  bp = __polyvec_ntt(bp);
  t = __polyvec_pointwise_acc(t, skpv, bp);
  t = _poly_invntt(t);

  mp = _poly_sub(mp, v, t);
  mp = __poly_reduce(mp);

  msgp, mp = _poly_tomsg_1(msgp, mp);

  return msgp;
}
```

jasmin avx2

Performance

- Reference implementation:
  - easier proof → slow
  - non-optimizing compiler
- AVX implementation (fully verified)
  - leave out one challenging routine ⏳
  - 100% penalty
- AVX implementation (fully optimized)
  - essentially matches unverified code
  - non-trivial parallelization



Implementation	**operation**	**Skylake**	**Haswell**	**Comet Lake**
C ref	keygen	200302	187172	184374
	encaps	251384	242424	235714
	decaps	287724	278160	272296
Jasmin ref	keygen	411676	394636	384948
	encaps	488904	471680	458640
	decaps	562426	534420	527266
C/asm AVX2	keygen	49572	47280	41682
	encaps	60018	62900	55956
	decaps	45854	47784	43906
Jasmin AVX2 (fully verified)	keygen	106578	96296	93244
	encaps	119308	111536	107474
	decaps	105336	98328	96564
Jasmin AVX2 (fully optimized)	keygen	50004	48800	45046
	encaps	65132	63988	59496
	decaps	50340	51444	48172

Jasmin needed to evolve

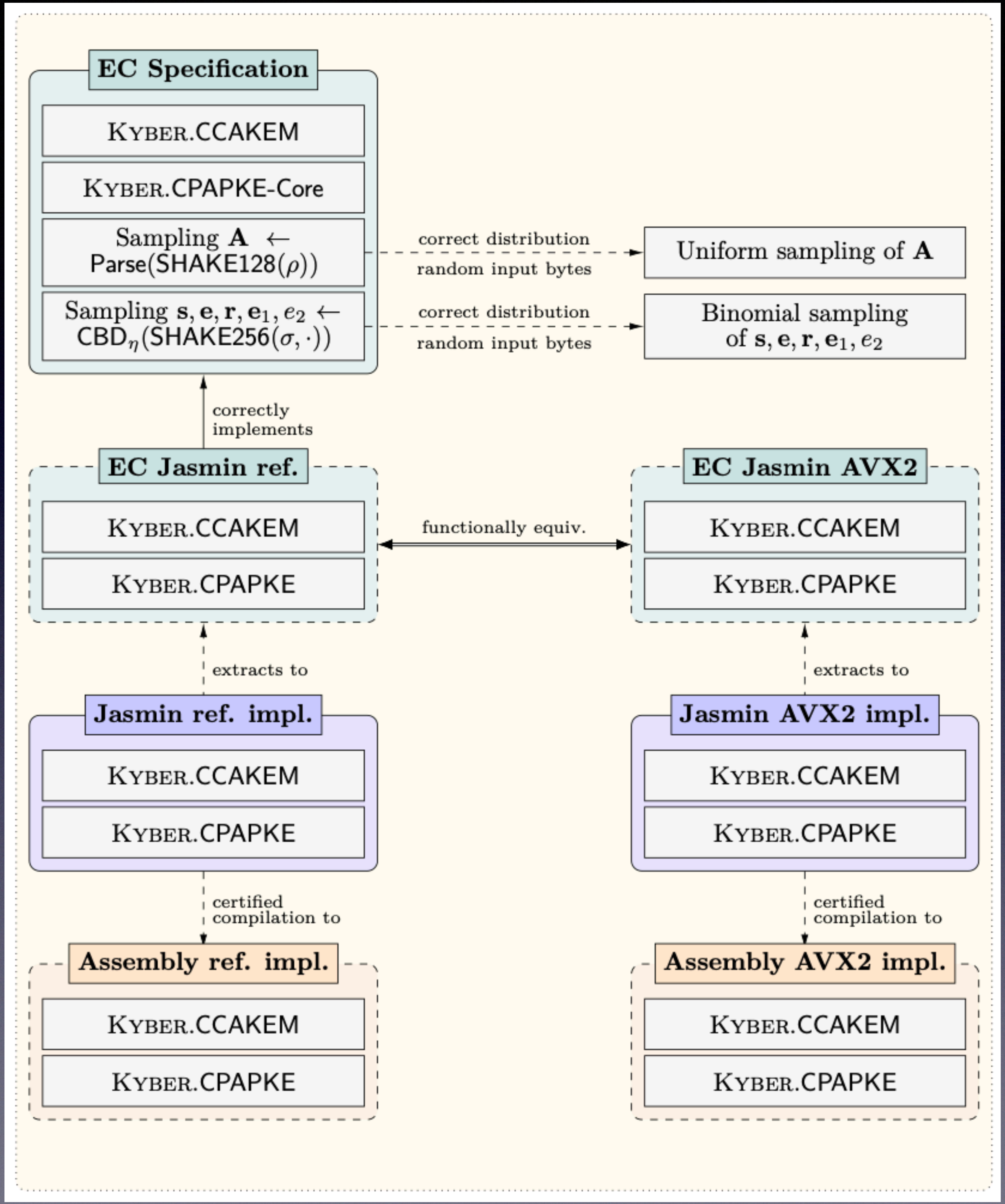
- First version of code: fully inlined: too large for compiler
- New features and extended proof for compiler (highlights):
  - local functions: new function call mechanism, smaller code
  - sub-arrays and implicit pointers to stack:
    - new stack management
    - sub-arrays: (slices of) stack can be passed "by reference"
  - random sampling: randombytes

Correctness Proof

- High-level view and top-level results
- Different approaches for ref and avx2
- Zoom-in on examples

High-level view

- Reference implementation:
    - Proof done first (along with security proof ⏳)
    - Most interesting challenges handled here:
        - Algebraic structure vs low-level implementations
        - NTT formalization and properties
        - Characterizing/validating SHA-3 usage
        - Correctness of sampling procedures
- AVX implementation
    - Unexpectedly challenging: hard to reuse proof above
    - A lot of effort for small additional scientific gain (?)
    - Huge practical gain (cf. benchmarks)



EC Specification

Kyber.CCAKEM

Kyber.CPAPKE-Core

Sampling $\mathbf{A} \leftarrow$ Parse(SHAKE128(ρ))

Sampling $\mathbf{s}, \mathbf{e}, \mathbf{r}, \mathbf{e}_1, e_2 \leftarrow$ CBD$_\eta$(SHAKE256($\sigma, \cdot$))

correct distribution

random input bytes

Uniform sampling of $\mathbf{A}$

correct distribution

random input bytes

Binomial sampling of $\mathbf{s}, \mathbf{e}, \mathbf{r}, \mathbf{e}_1, e_2$

correctly implements

EC Jasmin ref.

Kyber.CCAKEM

Kyber.CPAPKE

functionally equiv.

EC Jasmin AVX2

Kyber.CCAKEM

Kyber.CPAPKE

extracts to

extracts to

Jasmin ref. impl.

Kyber.CCAKEM

Kyber.CPAPKE

Jasmin AVX2 impl.

Kyber.CCAKEM

Kyber.CPAPKE

certified compilation to

certified compilation to

Assembly ref. impl.

Kyber.CCAKEM

Kyber.CPAPKE

Assembly AVX2 impl.

Kyber.CCAKEM

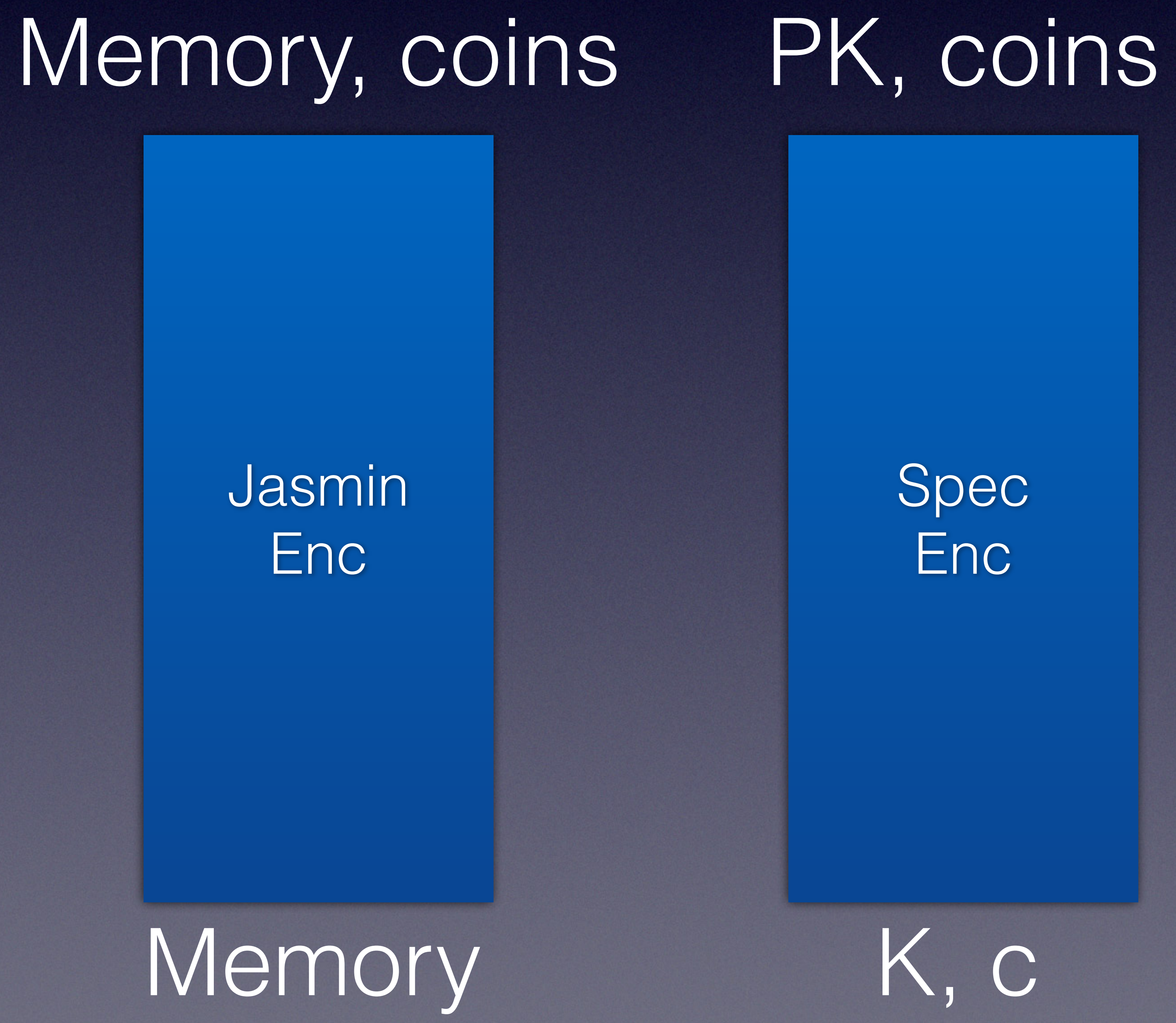
Kyber.CPAPKE

Top-level statements

Example Lemma: Kyber encapsulation is correctly implemented

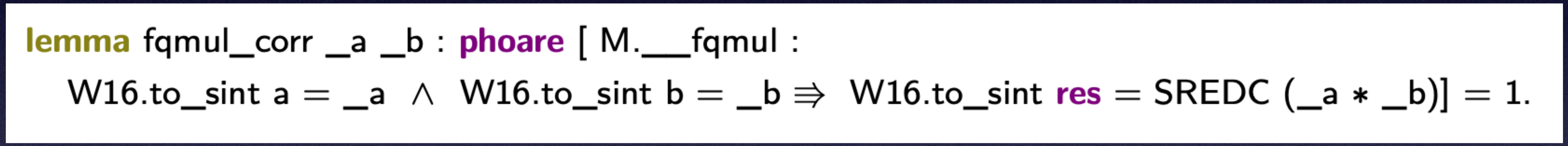
$\forall \mathsf{pkp}, \mathsf{ctp}, \mathsf{kp}, \mathsf{PK} :$
$\quad \mathsf{equiv} : \mathsf{JKem.enc} \sim \mathsf{KyberSpec.enc}$

pkp points to valid memory region $\wedge$
ctp points to valid memory region $\wedge$
kp points to valid memory region $\wedge$
ctp and kp point to disjoint memory regions $\wedge$
Starting holds PK
$\Longrightarrow$
Memory unchanged except in ctp, kp regions $\wedge$
Memory holds K and c

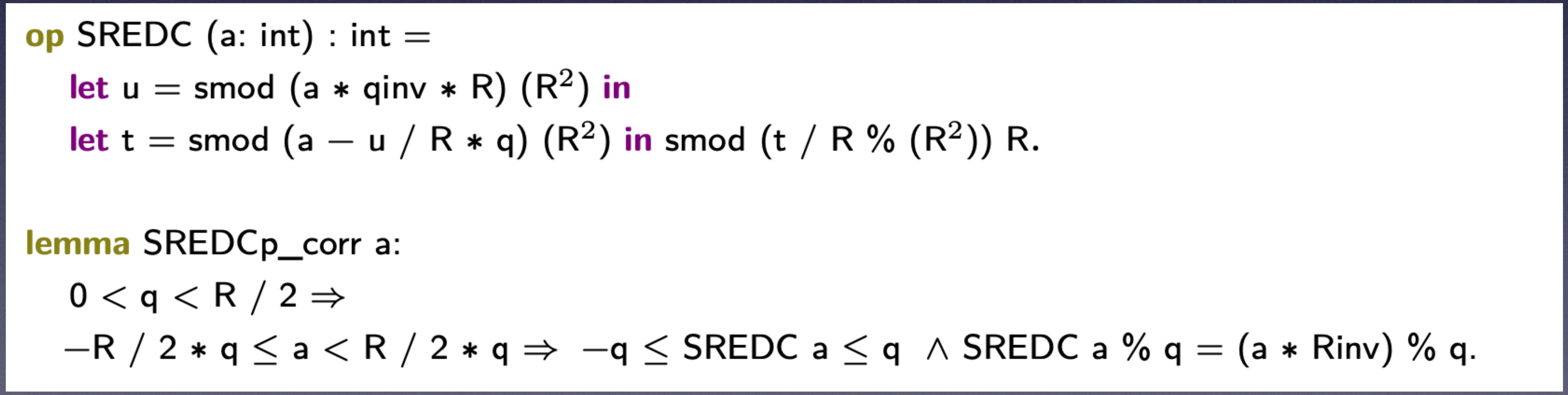


Verifying reference implementation

- Building results bottom up: field operations using Hoare logic



```
lemma fqmul_corr _a _b : phoare [ M.__fqmul :
   W16.to_sint a = _a ∧ W16.to_sint b = _b ⇒ W16.to_sint res = SREDC (_a ∗ _b)] = 1.
```

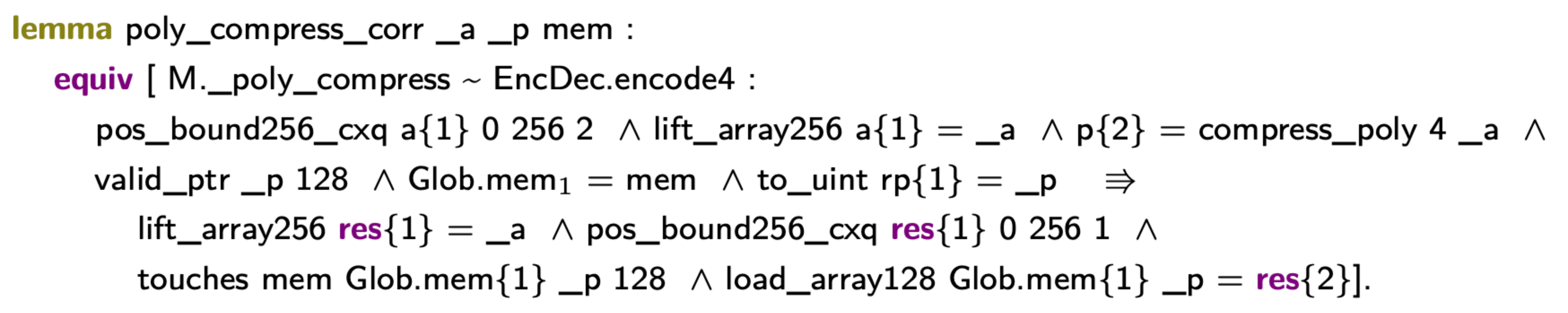


```
op SREDC (a: int) : int =
   let u = smod (a ∗ qinv ∗ R) (R^2) in
   let t = smod (a − u / R ∗ q) (R^2) in smod (t / R % (R^2)) R.

lemma SREDCp_corr a:
   0 < q < R / 2 ⇒
  −R / 2 ∗ q ≤ a < R / 2 ∗ q ⇒ −q ≤ SREDC a ≤ q ∧ SREDC a % q = (a ∗ Rinv) % q.
```

Spec in functional form comes with semantics and range properties.

Verifying reference implementation

- Building results bottom up: ring operations using relational logic

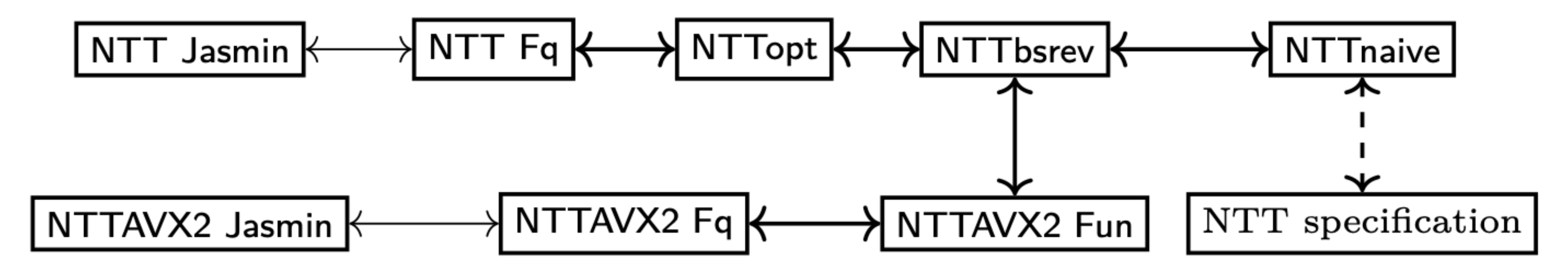


```
lemma poly_compress_corr _a _p mem :
  equiv [ M._poly_compress ~ EncDec.encode4 :
    pos_bound256_cxq a{1} 0 256 2  ∧ lift_array256 a{1} = _a  ∧ p{2} = compress_poly 4 _a  ∧
    valid_ptr _p 128  ∧ Glob.mem₁ = mem  ∧ to_uint rp{1} = _p    ⇒
      lift_array256 res{1} = _a  ∧ pos_bound256_cxq res{1} 0 256 1  ∧
      touches mem Glob.mem{1} _p 128  ∧ load_array128 Glob.mem{1} _p = res{2}].
```

Writing the spec in imperative form as an intermediate step makes proof easier

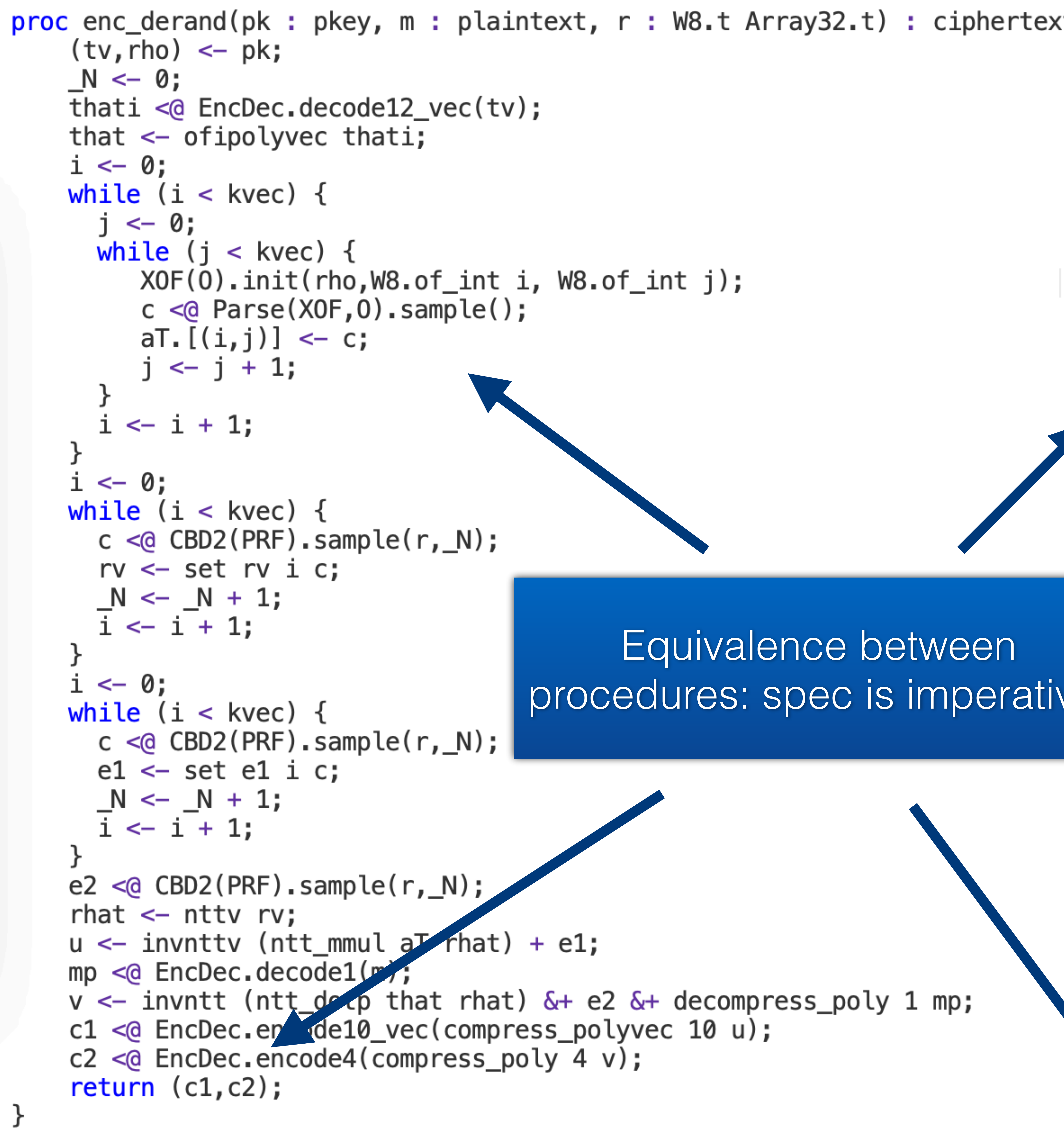
Verifying reference implementation

- One extreme case of imperative vs functional was NTT



- Huge semantic gap: mathematical view (properties) vs code
- Different loop structures and in-place computations
- Ref implementation completely different from avx2 implementation

At top level, equivalence follows from two types of results.

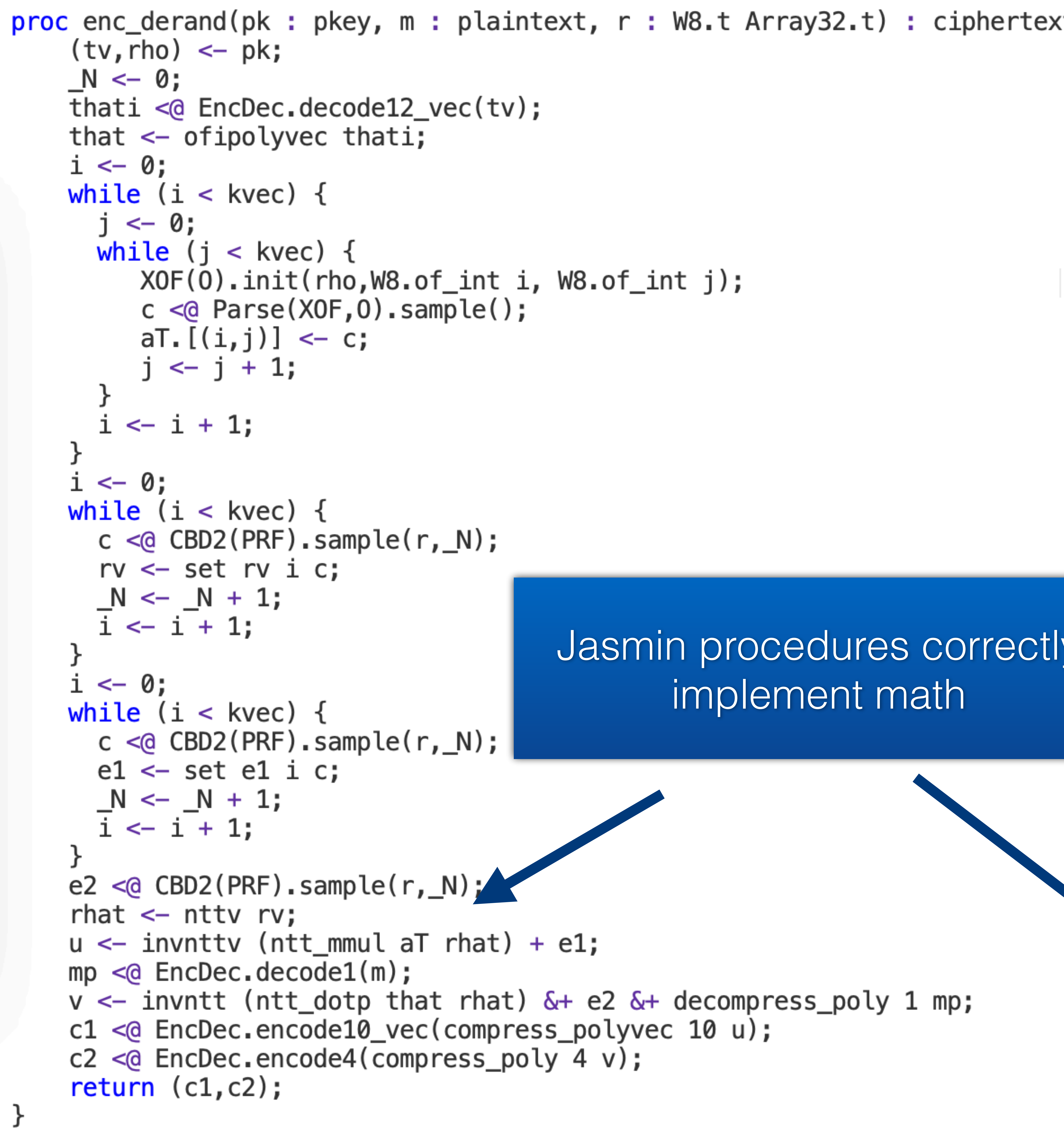


```
proc enc_derand(pk : pkey, m : plaintext, r : W8.t Array32.t) : ciphertext = {
    (tv,rho) <- pk;
    _N <- 0;
    thati <@ EncDec.decode12_vec(tv);
    that <- ofipolyvec thati;
    i <- 0;
    while (i < kvec) {
      j <- 0;
      while (j < kvec) {
         XOF(O).init(rho,W8.of_int i, W8.of_int j);
         c <@ Parse(XOF,O).sample();
         aT.[(i,j)] <- c;
         j <- j + 1;
      }
      i <- i + 1;
    }
    i <- 0;
    while (i < kvec) {
      c <@ CBD2(PRF).sample(r,_N);
      rv <- set rv i c;
      _N <- _N + 1;
      i <- i + 1;
    }
    i <- 0;
    while (i < kvec) {
      c <@ CBD2(PRF).sample(r,_N);
      e1 <- set e1 i c;
      _N <- _N + 1;
      i <- i + 1;
    }
    e2 <@ CBD2(PRF).sample(r,_N);
    rhat <- nttv rv;
    u <- invnttv (ntt_mmul aT rhat) + e1;
    mp <@ EncDec.decode1(m);
    v <- invntt (ntt_dotp that rhat) &+ e2 &+ decompress_poly 1 mp;
    c1 <@ EncDec.encode10_vec(compress_polyvec 10 u);
    c2 <@ EncDec.encode4(compress_poly 4 v);
    return (c1,c2);
}
```

Equivalence between procedures: spec is imperative.

```
inline
fn __indcpa_enc(stack u64 sctp, reg ptr u8[32] msgp,
                reg u64 pkp, reg ptr u8[KYBER_SYMBYTES] noiseseed)
{
  pkpv = __polyvec_frombytes(pkp);
  k = _i_poly_frommsg(k, msgp);
  aat = __gen_matrix(publicseed, 1);

  nonce = 0; sp[0:KYBER_N] = _poly_getnoise(sp[0:KYBER_N], noiseseed, nonce);
  nonce = 1; sp[KYBER_N:KYBER_N] = _poly_getnoise(sp[KYBER_N:KYBER_N], noiseseed, nonce);
  nonce = 2; sp[2*KYBER_N:KYBER_N] = _poly_getnoise(sp[2*KYBER_N:KYBER_N], noiseseed, nonce);

  nonce = 3; ep[0:KYBER_N] = _poly_getnoise(ep[0:KYBER_N], noiseseed, nonce);
  nonce = 4; ep[KYBER_N:KYBER_N] = _poly_getnoise(ep[KYBER_N:KYBER_N], noiseseed, nonce);
  nonce = 5; ep[2*KYBER_N:KYBER_N] = _poly_getnoise(ep[2*KYBER_N:KYBER_N], noiseseed, nonce);

  nonce = 6; epp = _poly_getnoise(epp, noiseseed, nonce);

  sp = __polyvec_ntt(sp);
  bp[0:KYBER_N] = __polyvec_pointwise_acc(aat[0:KYBER_VECN], sp);
  bp[KYBER_N:KYBER_N]= __polyvec_pointwise_acc(aat[KYBER_VECN:KYBER_VECN], sp);
  bp[2*KYBER_N:KYBER_N] = __polyvec_pointwise_acc(aat[2*KYBER_VECN:KYBER_VECN], sp);

  v = __polyvec_pointwise_acc(pkpv, sp);
  bp = __polyvec_invntt(bp);
  v = _poly_invntt(v);

  bp = __polyvec_add2(bp, ep);
  v = _poly_add2(v, epp);
  v = _poly_add2(v, k);
  bp = __polyvec_reduce(bp);
  v  = __poly_reduce(v);

  ctp = sctp;
  __polyvec_compress(ctp, bp);
  ctp += KYBER_POLYVECCOMPRESSEDBYTES;
  v = _poly_compress(ctp, v);
}
```

At top level, equivalence follows from two types of results.



```
proc enc_derand(pk : pkey, m : plaintext, r : W8.t Array32.t) : ciphertext = {
    (tv,rho) <- pk;
    _N <- 0;
    thati <@ EncDec.decode12_vec(tv);
    that <- ofipolyvec thati;
    i <- 0;
    while (i < kvec) {
      j <- 0;
      while (j < kvec) {
         XOF(O).init(rho,W8.of_int i, W8.of_int j);
         c <@ Parse(XOF,O).sample();
         aT.[(i,j)] <- c;
         j <- j + 1;
      }
      i <- i + 1;
    }
    i <- 0;
    while (i < kvec) {
      c <@ CBD2(PRF).sample(r,_N);
      rv <- set rv i c;
      _N <- _N + 1;
      i <- i + 1;
    }
    i <- 0;
    while (i < kvec) {
      c <@ CBD2(PRF).sample(r,_N);
      e1 <- set e1 i c;
      _N <- _N + 1;
      i <- i + 1;
    }
    e2 <@ CBD2(PRF).sample(r,_N);
    rhat <- nttv rv;
    u <- invnttv (ntt_mmul aT rhat) + e1;
    mp <@ EncDec.decode1(m);
    v <- invntt (ntt_dotp that rhat) &+ e2 &+ decompress_poly 1 mp;
    c1 <@ EncDec.encode10_vec(compress_polyvec 10 u);
    c2 <@ EncDec.encode4(compress_poly 4 v);
    return (c1,c2);
}
```

Jasmin procedures correctly implement math

```
inline
fn __indcpa_enc(stack u64 sctp, reg ptr u8[32] msgp,
                reg u64 pkp, reg ptr u8[KYBER_SYMBYTES] noiseseed)
{
  pkpv = __polyvec_frombytes(pkp);
  k = _i_poly_frommsg(k, msgp);
  aat = __gen_matrix(publicseed, 1);

  nonce = 0; sp[0:KYBER_N] = _poly_getnoise(sp[0:KYBER_N], noiseseed, nonce);
  nonce = 1; sp[KYBER_N:KYBER_N] = _poly_getnoise(sp[KYBER_N:KYBER_N], noiseseed, nonce);
  nonce = 2; sp[2*KYBER_N:KYBER_N] = _poly_getnoise(sp[2*KYBER_N:KYBER_N], noiseseed, nonce);

  nonce = 3; ep[0:KYBER_N] = _poly_getnoise(ep[0:KYBER_N], noiseseed, nonce);
    nce = 4; ep[KYBER_N:KYBER_N] = _poly_getnoise(ep[KYBER_N:KYBER_N], noiseseed, nonce);
    nce = 5; ep[2*KYBER_N:KYBER_N] = _poly_getnoise(ep[2*KYBER_N:KYBER_N], noiseseed, nonce);

    nce = 6; epp = _poly_getnoise(epp, noiseseed, nonce);

     = __polyvec_ntt(sp);
  bp[0:KYBER_N] = __polyvec_pointwise_acc(aat[0:KYBER_VECN], sp);
  bp[KYBER_N:KYBER_N]= __polyvec_pointwise_acc(aat[KYBER_VECN:KYBER_VECN], sp);
  bp[2*KYBER_N:KYBER_N] = __polyvec_pointwise_acc(aat[2*KYBER_VECN:KYBER_VECN], sp);

  v = __polyvec_pointwise_acc(pkpv, sp);
  bp = __polyvec_invntt(bp);
  v = _poly_invntt(v);

  bp = __polyvec_add2(bp, ep);
  v = _poly_add2(v, epp);
  v = _poly_add2(v, k);
  bp = __polyvec_reduce(bp);
  v  = __poly_reduce(v);

  ctp = sctp;
  __polyvec_compress(ctp, bp);
  ctp += KYBER_POLYVECCOMPRESSEDBYTES;
  v = _poly_compress(ctp, v);
}
```

AVX2 Implementation

- Different instruction sequences to compute same result (e.g., compression)
    - no alternative to proving additional results for lower-level routines
- Computations done in different order (unrelated control flow)
    - very little high-level structure (e.g., NTT computation)
    - no alternative to proving additional results for NTT procedures
- Totally different approach to some procedures (e.g., rejection sampling matrix A)
    - aggressive optimisations: different reasoning about sampling semantics

AVX2 Implementation

Once we have intermediate results that match AVX2 procedures to ref procedures

- n)
  - no alternative to proving additional results for lower-level routines
- Computations done in different order (unrelated control flow)

High-level equivalence proofs can be reused:

$$\text{AVX2} \equiv \text{Ref} \Rightarrow \text{Ref} \equiv \text{Spec} \Rightarrow \text{AVX2} \equiv \text{Spec}$$

  - aggressive optimisations: different reasoning about sampling semantics

EasyCrypt needed extending

- A lot of extensions to standard library
  - polynomial arithmetic, ring quotients, bit-vector manipulations, etc.
- Automatic inference of functional specs
  - no need to prove imperative code implements operator
- Library for dealing with nested loops

Conclusions and Future Work

- Lessons Learned
- Ongoing work
- Long(er)-term goals

Lessons learned

- Three years!
  - Improve tools
  - Train people
  - Availability/coordination
- Still if we started now
  - Significant investment

Lessons learned

- Three years!
  - Improve
  - Train people
  - Availability/coordination
- Still if we started now
  - Significant investment

We need more automation!

And a stable team of developers!

Investment returns

- Non-ambiguous specification: we can formally reason about a future standard
  - Prove properties of spec: does paper proof apply?
  - Implementation inherits properties
  - Connection to security proof, e.g.:
    - SHA-3, SHAKE usage
    - Assumed security properties
    - E.g., model as independent RO?
    - E.g. model as PRF, PRG?

Investment returns

- Bugs might not be caught by testing:
  - Timing attacks
  - Spectre v1
  - Rare algebraic errors
  - Sampling from incorrect distributions
- Proof requires deep insights:
  - Can (has) lead to additional speed-ups

Future/Ongoing work

- Increase automation in verification framework
- libjade:
  - Proofs for other (post-quantum) schemes (and Kyber avx2)
  - Other architectures, namely ARM (is proof effort amortized?)
  - Getting code out there:
    libraries, bindings to other languages, real-world applications
  - Move to low-level protocols (key exchange, authentication, etc.)