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.

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. We conclude with a summary of open challenges 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 design, 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. One shows the importance of having security and provable correctness for modular constructions of designs, 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

Interactively in a Zulip server

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.

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 Compiler

Safety check  CT check  Spectre v1 check  certified compilation

EasyCrypt

Functional correctness proof  Security proof

Asm code (ref)  Asm code (avx)  Asm code (avx2)

Algorithm spec  Security model  Claims

Under the hood  Use  Inspect
Formal Verification Approach

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

- Jasmin Code
  - Safety check
  - CT check
- asm Code
- Algorithm spec
- Machine-checked in EasyCrypt

- Functional correctness I
- No timing leakage (*)
- No leakage due to Spectre v1 (*)

(*) in a formally defined (abstract) leakage model
Formal verification goal

- Jasmin Code
  - Safety check
  - CT check
  - certified compilation
  - Spectre v1 Check
  - asm Code

- Algorithm spec
  - crypto proof
  - Security model

- Functional correctness
  - No timing leakage (*)
  - No leakage due to Spectre v1 (*)

- Certification
  - Implementation security
  - Compliance/Interoperability

- e.g., Kyber asm behaves like Kyber jasmin

(*) in a formally defined (abstract) leakage model
Formal verification goal

This talk!

Jasmin Code

Functional Correctness (interactive)

Implementation security

Algorithm spec

Machine-checked in EasyCrypt

Safety check

CT check

Certified compilation

Spectre v1 Check

asm Code

No timing leakage (*)

No leakage due to Spectre v1 (*)

Functional correctness (automatic)

Security model

Crypto proof

Security compliance/Interoperability

This talk! depends on Spec

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

e.g., Kyber asm behaves like Kyber jasmin

Machine-checked in EasyCrypt

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

- Jasmin Code
  - Safety check
  - CT check
  - certified compilation
  - Spectre v1 Check

- Functional Correctness (interactive)
  - implementation security
  - compliance/Interoperability

- Algorithm spec
  - crypto proof
  - Security model
  - Machine-checked in EasyCrypt

- Other specs? (e.g., HACSpec)
- Standard?

- Functional correctness (automatic)
  - No timing leakage (*)
  - No leakage due to Spectre v1 (*)

- e.g., Kyber asm behaves like Kyber jasmin

(*) 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

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

- 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

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

- 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

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

Programmer responsible for all spilling

Compilation breaks if register assignment not found.
Jasmin: per arch instruction set

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

```rust
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;
}
```

```rust
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 >>= 15;
        b &= KYBER_Q;
        t = b;
        rp[(int)i] = t;
        i += 1;
    }
    return rp;
}
```
Jasmin: per arch instruction set

- 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

- 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

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

```haskell
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;
  }
}.
```

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

lemma relate : ∀ .x .y .v2, hoare[M.f : arg=(x,y) ∧ M.v2 = .v2 → res=x + y ∧ M.v2 = .v2].
Hoare logic

Your usual Hoare triple based on two predicates

Precondition: assumed in starting state
Postcondition: ensured in final state

In this work: prove that procedures implement convenient functional specs

E.g., Jasmin code implements inner product correctly

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;
}.

Lemma relate : ∀ x y v2, Hoare[M.1 : arg=(x,y) ∧ M.v2 = v2 ⇒ res=x + y ∧ M.v2=v2].
Relational Hoare logic

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

```plaintext
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;
    }
}.
```

equiv relate x : M.f ∼ M.g : arg{1}=(x,x) ∧ arg{2} = x → ={res}.
Relational Hoare logic

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

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

```
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;
    }
}.
```

```
equiv relate _x : M.f ~ M.g : arg{1}=(_,_x) ∧ arg{2} = _x ⇒ ={res}.
```
Relational Hoare logic

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

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

spec vs implementation

```text
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;
  }
};
```

equiv relate \( \forall x : M.f \sim M.g : \arg{1} = (x, x) \land \arg{2} = x \implies \{\text{res}\} \).
Relational Hoare logic

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

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

implementation vs optimized implementation
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

```verbatim
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(FqD).
rewrite (Ring.IntID.expr5 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
- \( \mathbb{F}_q \): field, integers modulo \( q \), type of coefficients
- \( 
\mathbb{R}_q : \) ring of polynomials modulo \((X^{256} + 1)\) over \( \mathbb{F}_q \)

**Bold lower caps:** col vectors of size \( k \) over \( \mathbb{R}_q \)

**Bold upper caps:** \( k \times k \) matrix over \( \mathbb{R}_q \)

- \( s, e, r, e_1, e_2 \) small norm: each coeff. Binomial distr.
- A coeffs. sampled uniformly from \( \mathbb{F}_q \)
- Multiplications in \( \mathbb{R}_q \) done in NTT domain
- Enc/Dec: encoding and decoding operations

---

**Kyber.CPAPKE: LPR encryption or “Noisy ElGamal”**

\[
\begin{align*}
\text{s, } e \leftarrow \chi \\
sk = s, \ pk = t = As + e
\end{align*}
\]

\[
\begin{align*}
\text{r, } e_1, e_2 \leftarrow \chi \\
\text{u } &\leftarrow A^T r + e_1 \\
\nu &\leftarrow t^T r + e_2 + \text{Enc}(m) \\
c &\leftarrow (u, \nu)
\end{align*}
\]

\[
m = \text{Dec}(\nu - s^T u)
\]

---

**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

---

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.
```

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

```
lemma invntttK : cancel ntt invnttt.
```

```
Compress_q(x, d) = \lceil (2^d/q) \cdot x \rceil \mod 2^d
```

```
x' = Decompress_q(Compress_q(x, d), d)
|x' - x \mod \pm q| \leq B_q := \left\lceil \frac{q}{2^{d+1}} \right\rceil
```

```
ntt(p : poly) = 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)
```

```
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 br(i)+1)j},
```

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

```haskell
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$_{\eta}$ : $B^{64\eta} \rightarrow R_q$

**Input:** Byte array $B = (b_0, b_1, \ldots, b_{64\eta-1}) \in B^{64\eta}$
**Output:** Polynomial $f \in R_q$

$(\beta_0, \ldots, \beta_{512\eta-1}) := \text{BytesToBits}(B)$

for $i$ from 0 to 255 do
  $a := \sum_{j=0}^{\eta-1} \beta_{2i\eta+j}$
  $b := \sum_{j=0}^{\eta-1} \beta_{2i\eta+j}$
  $f_i := a - b$
end for

return $f_0 + f_1X + f_2X^2 + \cdots + f_{255}X^{255}$

**equiv** CBD2rnd_equiv:

CBD2rnd.sample_real $\sim$ CBD2rnd.sample_ideal:

true $\implies$ ={res}.

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

```plaintext
proc enc_derand(pk : pkey, m : plaintext, r : WB.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(0).init(rho, WB.of_int i, WB.of_int j);
            c <- Parse(XOF(0).sample());
            aT.[(i,j)] <- c;
            j <- j + 1;
        }
        i <- i + 1;
    }
    i <- 0;
    while (i < kvec) {
        c <- CBD2(PR).sample(r,N);
        rv <- set rv i c;
        N <- N + 1;
        i <- i + 1;
    }
    i <- 0;
    while (i < kvec) {
        c <- CBD2(PR).sample(r,N);
        e1 <- set e1 i c;
        N <- N + 1;
        i <- i + 1;
    }
    e2 <- CBD2(PR).sample(r,N);
    rhat <- nttv rv;
    u <- invnttv (ntt_mul aT rhat) + e1;
    mp <- EncDec.decode1(m);
    v <- invntt (ntt_dot prod 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 $pk \in B^{12 \cdot k \cdot n/8+32}$
Input: Message $m \in B^{32}$
Input: Random coins $r \in B^{32}$

Output: Ciphertext $c \in B^{d_u \cdot k \cdot n/8+d_v \cdot n/8}$

1: $N := 0$
2: $i := \text{Decode}_{12}(pk)$
3: $\rho := pk + 12 \cdot k \cdot n/8$
4: for $i$ from 0 to $k-1$ do
5:     for $j$ from 0 to $k-1$ do
6:         $\hat{A}^T[i][j] := \text{Parse}(XOF(\rho, i, j))$
7:     end for
8: end for
9: for $i$ from 0 to $k-1$ do
10:    $r[i] := \text{CBD}_{\eta_1}(\text{PRF}(r,N))$
11:    $N := N + 1$
12: end for
13: for $i$ from 0 to $k-1$ do
14:    $e_1[i] := \text{CBD}_{\eta_2}(\text{PRF}(r,N))$
15:    $N := N + 1$
16: end for
17: $e_2 := \text{CBD}_{\eta_2}(\text{PRF}(r,N))$
18: $\hat{r} := \text{NTT}(r)$
19: $u := \text{NTT}^{-1}(\hat{A}^T \circ \hat{r}) + e_1$
20: $v := \text{NTT}^{-1}(\hat{r}^T \circ \hat{r}) + e_2 + \text{Decompress}_{\eta}(\text{Decode}_{1}(m), 1)$
21: $c_1 := \text{Encode}_{d_u}(\text{Compress}_{\eta}(u, d_u))$
22: $c_2 := \text{Encode}_{d_v}(\text{Compress}_{\eta}(v, d_v))$
23: return $c = (c_1 \parallel c_2)$
Jasmin Implementation

- Structure of the code
- Performance
- Snippets/examples
Structure of Jasmin code

- reference
- params.jinc
- indcpa.jinc
- verify.jinc
- polyvec.jinc
- gen_matrix.jinc
- poly.jinc
- consts.jinc
- reduce.jinc
- fips202.jinc
  - SHA-3 code
- kem.jinc

SHA-3 code
Structure of Jasmin code
Structure of Jasmin code

- kem.jinc
- indcpa.jinc
- verify.jinc
- params.jinc
- polyvec.jinc
- gen_matrix.jinc
- poly.jinc
- shuffle.jinc
- consts.jinc
- reduce.jinc
- fips202_4x.jinc
- SHA-3 code
- fips202.jinc
- SHA-3 code

reference
extra in avx2
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;
}

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;
Structure of Jasmin code

```c
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);
}
```

```c
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;
}
```

```c
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;
}
```
Structure of Jasmin code

- reference
- extra in avx2
- kem.jinc
- indcpa.jinc
- verify.jinc
- params.jinc
- polyvec.jinc
- poly.jinc
- shuffle.jinc
- gen_matrix.jinc
- reduce.jinc
- const.sinc
- fips202.jinc
- SHA-3 code
- fips202_4x.jinc
- SHA-3 code
Structure of Jasmin 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);
}

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_POLYVECCOMPRESSIONBYTES;
    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;
}

inline fn __indcpa_dec(reg ptr u8[KYBER_INDCPA_MSGBYTES] msgp,
        [reg u64 ctp, reg u64 skp] -> reg ptr u8[KYBER_INDCPA_MSGBYTES])
{
    stack u16[KYBER_N] t v mp;
    stack u16[KYBER_VECN] bp skpv;
    bp = __polyvec_decompress(ctp);
    ctp += KYBER_POLYVECCOMPRESSIONBYTES;
    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;
}
### Performance

- Reference implementation:
  - easier proof $\rightarrow$ 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

<table>
<thead>
<tr>
<th>Implementation</th>
<th>operation</th>
<th>Skylake</th>
<th>Haswell</th>
<th>Comet Lake</th>
</tr>
</thead>
<tbody>
<tr>
<td>C ref</td>
<td>keygen</td>
<td>200302</td>
<td>187172</td>
<td>184374</td>
</tr>
<tr>
<td></td>
<td>encaps</td>
<td>251384</td>
<td>242424</td>
<td>235714</td>
</tr>
<tr>
<td></td>
<td>decaps</td>
<td>287724</td>
<td>278160</td>
<td>272296</td>
</tr>
<tr>
<td>Jasmin ref</td>
<td>keygen</td>
<td>411676</td>
<td>394636</td>
<td>384948</td>
</tr>
<tr>
<td></td>
<td>encaps</td>
<td>488904</td>
<td>471680</td>
<td>458640</td>
</tr>
<tr>
<td></td>
<td>decaps</td>
<td>562426</td>
<td>534420</td>
<td>527266</td>
</tr>
<tr>
<td>C/asm AVX2</td>
<td>keygen</td>
<td>49572</td>
<td>47280</td>
<td>41682</td>
</tr>
<tr>
<td></td>
<td>encaps</td>
<td>60018</td>
<td>62900</td>
<td>55956</td>
</tr>
<tr>
<td></td>
<td>decaps</td>
<td>45854</td>
<td>47784</td>
<td>43906</td>
</tr>
<tr>
<td>Jasmin AVX2</td>
<td>keygen</td>
<td>106578</td>
<td>96296</td>
<td>93244</td>
</tr>
<tr>
<td>(fully verified)</td>
<td>encaps</td>
<td>119308</td>
<td>111536</td>
<td>107474</td>
</tr>
<tr>
<td></td>
<td>decaps</td>
<td>105336</td>
<td>98328</td>
<td>96564</td>
</tr>
<tr>
<td>Jasmin AVX2</td>
<td>keygen</td>
<td>50004</td>
<td>48800</td>
<td>45046</td>
</tr>
<tr>
<td>(fully optimized)</td>
<td>encaps</td>
<td>65132</td>
<td>63988</td>
<td>59496</td>
</tr>
<tr>
<td></td>
<td>decaps</td>
<td>50340</td>
<td>51444</td>
<td>48172</td>
</tr>
</tbody>
</table>
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)
Example Lemma: Kyber encapsulation is correctly implemented

∀pkp, ctp, kp, PK :
  equiv : JKem.enc ∼ KyberSpec.enc

pkp points to valid memory region ∧
ctp points to valid memory region ∧
kp points to valid memory region ∧
ctp and kp point to disjoint memory regions ∧
Starting holds PK
⇒
Memory unchanged except in ctp, kp regions ∧
Memory holds K and c
Verifying reference implementation

• Building results bottom up: field operations using Hoare logic

```
lemma fmul_corr__a__b : phoare [ M.___fmul :
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.

Equivalence between procedures: spec is imperative.
At top level, equivalence follows from two types of results.

Jasmin procedures correctly implement math.
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

High-level equivalence proofs can be reused:

\[ AVX2 \equiv \text{Ref} \Rightarrow \text{Ref} \equiv \text{Spec} \Rightarrow AVX2 \equiv \text{Spec} \]
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 tools

• 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.)