Using Formally Verified Post-Quantum Algorithms at Scale

Karthikeyan Bhargavan, **Andres Erbsen**, **Lucas Franceschino**, Franziskus Kiefer, Thyla van der Merwe

Real World Cryptography Symposium
March 27, 2025

CRYSPEN Google

A Collaboration is Born

Why PQC?

- Quantum Computers soon? Transition now!
  - Attack: store now, decrypt later
- Industry standards, government customers
  - By ~2026!
- 1st Priority: Key Exchange in SSH/TLS/…
- Next: digital signatures
- Many products (some OSS), industry-wide effort → Open Source

Why Verify Lattice-Crypto Implementations?

- Goal: **no implementation vulnerabilities** in **optimized** code
- Experience from Elliptic-Curve Cryptography
  - Auditing code is important but challenging
  - Subtle bugs missed in high-profile implementations
- Simpler than ECC? (No carry chains, standard representations)
  - Yes, in reference implementations
- Tricky optimizations: vectorization, deferred reductions, decoding

The Technical Details

FIPS 203

Federal Information Processing Standards Publication

Module-Lattice-Based Key-Encapsulation Mechanism Standard

Category: Computer Security **Subcategory: Cryptography**

Information Technology Laboratory
National Institute of Standards and Technology
Gaithersburg, MD 20899-8900

This publication is available free of charge from:
https://doi.org/10.6028/NIST.FIPS.203

Published August 13, 2024

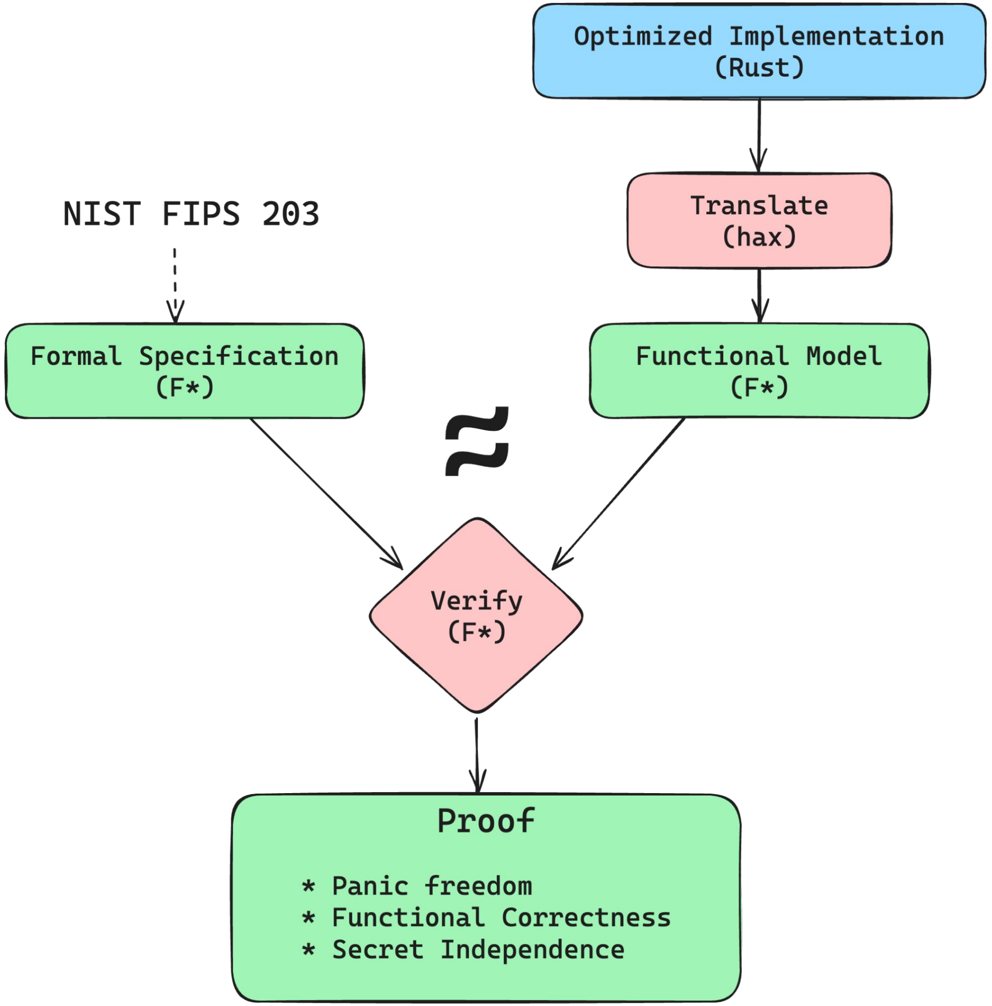
We implement and verify this standard.

Implementing ML-KEM in Rust

- Pure Rust code: 16 KLOC
- Optimized for multiple platforms
  - Portable + AVX2 + AArch64
  - 2 KLOC for SIMD optimizations (using intrinsics)
- Easy to integrate and deploy
  - Cargo crate: libcrux-ml-kem
  - PQCA's official Rust implementation

Mathematics	**Low-Level Formats**	**Algorithms**	**High-Level APIs**
Field, polynomial, matrix	(de)serialization	Sampling, IND-CPA, IND-CCA	ML-KEM 512/768/1024
3k lines	3k lines	6k lines	4k lines

Verifying crypto code written in Rust using hax and F*



Writing Crypto Code in Rust

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

Signed Barrett Reduction: with modulus **3329**
(in constant time, so cannot directly use **%**)

Specifying Correctness

```
#[requires(input <= 0x4_000_000 && input >= -0x4_000_000)]
#[ensures(|result| result <= 3328 && result >= -3328 &&
             modulo(result, 3329) == modulo(input, 3329))]
pub fn barrett_reduce(input: i32) -> i32 {
    ...
}
```

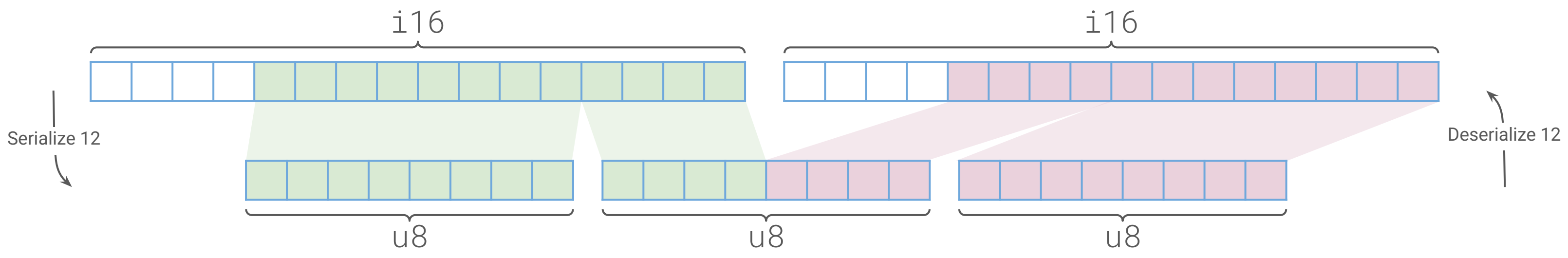
Expected behaviour: compute a signed representative of the input field element (modulo 3329)

Preventing Panics in Rust Code

```
#[requires(input <= 0x4_000_000 && input >= -0x4_000_000)]
#[ensures(...)]
pub fn barrett_reduce(input: i32) -> i32 {
    let t = (input as i64 * 20159) + (0x4_000_000 >> 1);
    let quotient = (t >> 26) as i32;
    let result = input - (quotient * 3329);
    result
}
```

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

Verifying (De-)Serialization Automatically



24 hand-optimized variants!

12 bits per integer	11 bits per integer	10 bits per integer	5 bits per integer	4 bits per integer	1 bits per integer	`serialize` `deserialize`	`portable` `avx2`
6 variants						2 variants	2 variants

A new F* tactic that can prove every variant automatically!

Enforcing Secret Independence

Type-based static analysis enforces a “constant-time” discipline

- **arithmetic operations** with input-dependent timing (e.g. division) over secret integers
- **comparison** over secret values
- **branching** over secret values
- array or vector **accesses** at secret indices

Prevents a large class of remote timing attacks (at source level).

Does not prevent compiler-induced leaks, micro-architectural attacks, …

KyberSlash: a new timing vulnerability

```
void poly_tomsg(uint8_t msg[KYBER_INDCPA_MSGBYTES], const poly *a)
{
  unsigned int i,j;
  uint16_t t;

  for(i=0;i<KYBER_N/8;i++) {
    msg[i] = 0;
    for(j=0;j<8;j++) {
      t  = a->coeffs[8*i+j];
      t += ((int16_t)t >> 15) & KYBER_Q;
      t  = (((t << 1) + KYBER_Q/2)/KYBER_Q) & 1;
      msg[i] |= t << j;
    }
  }
}
```

Bug found in our Rust code during formal verification

Bug also present in PQ-Crystals, PQ-Clean, … (used in production)

KyberSlash: Exploiting secret-dependent division timings in Kyber Implementations. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, *2025*(2), 209-234. Bernstein, D. J., Bhargavan, K., Bhasin, S., Chattopadhyay, A., Chia, T. K., Kannwischer, M. J., Kiefer, F., Paiva, T. B., Ravi, P., & Tamvada, G.

Scaling the Proof Effort

- Full formal verification of a large code-base
  - Source Rust code: 16 KLOC
  - Generated F* model: 28 KLOC (Portable + AVX2)
- Multiple automation strategies
  - SMT-based automation for low-level mathematics
  - Tactic-based automation for serialization
  - Type-based secret independence analysis
- Still needs many manual F* proofs + annotations for the full proof

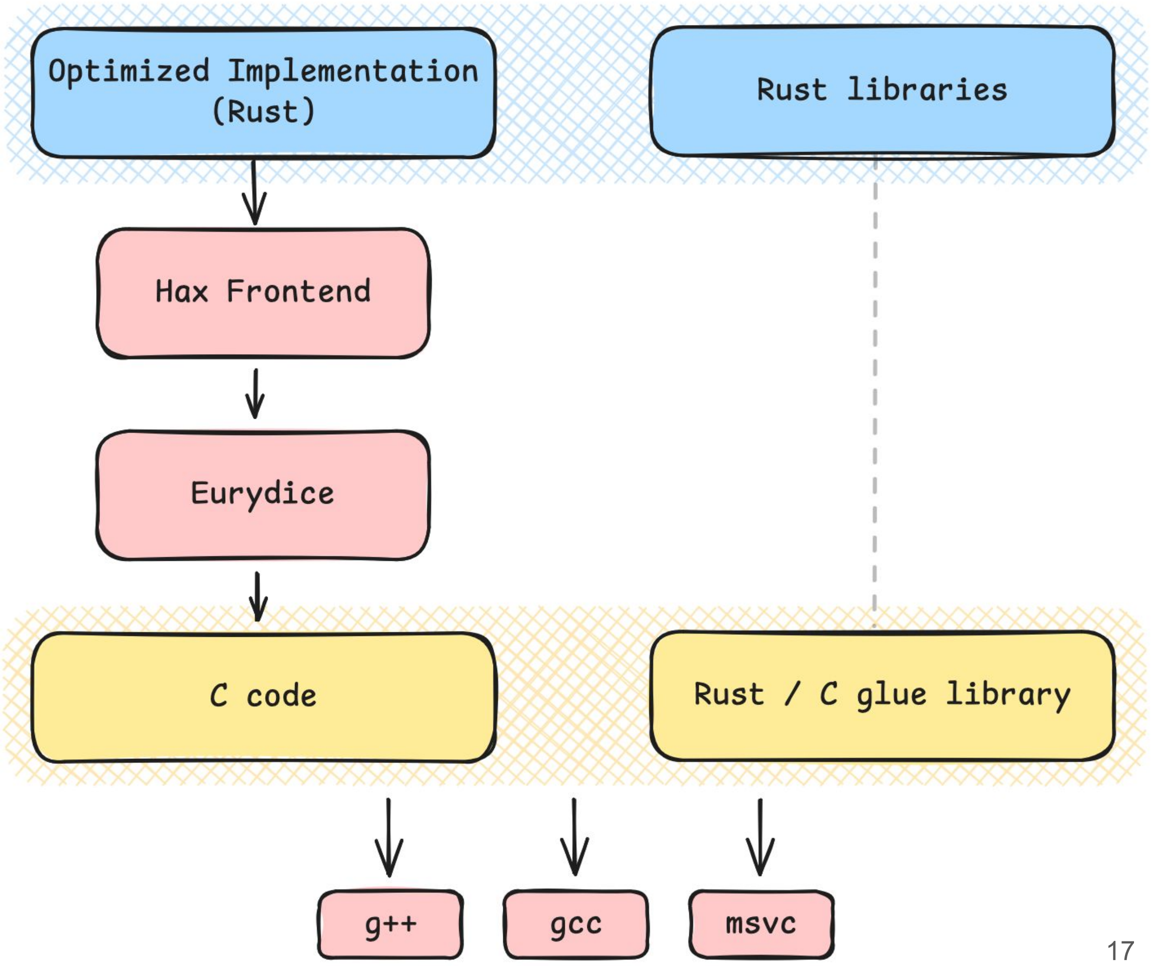
Mathematics	Low-Level Formats	Algorithms	High-Level APIs
Field, polynomial, matrix	(de)serialization	Sampling, IND-CPA, IND-CCA	ML-KEM 512/768/1024
6k lines of F*	5k lines of F*	6k lines of F*	4k lines of F*

Verified PQC at Scale

Compiling verified Rust to C code

Many mainstream crypto libraries need C code

NSS, BoringSSL, OpenSSH, OpenSSL, …



Integration Challenges

- C code size is larger than Rust
  - From monomorphizing ML-KEM variants from Rust
  - ~40KB optimized for speed
- Match existing APIs in the crypto library
  - Opaque secret keys, Alignment, Strict aliasing
- C++ toolchain compatibility (yes, even iOS, MSVC, ARM, bigendian…)
- Scale: ~100 build configurations, and evolving

Maintainability and Performance

- Establishing speciality tooling
  - Change workflow: modify Rust code, re-prove, re-generate C
  - Review specs, not code – but computer-check proofs!
  - Continuous integration for tools (ARM/Intel × Debian/MacOS)
  - Long-term support from Cryspen
- AVX2: ~2x faster than BoringSSL reference implementation
  - A great argument for at-scale deployment!

Takeaways

- PQC is coming, verification is important
  - Demonstrated with ML-KEM and KyberSlash
  - Deployments in OpenSSH, NSS, PQCA, Signal, Dropbear
- Many challenges need solving between
  - Formally verified fast code
  - At-scale deployment
- Next up: ML-DSA

Try our Rust or C code today!