Formal Verification of ML-KEM: Portable and AVX2

In this document, we describe the ongoing effort to formally verify the open-source implementation of ML-KEM being developed by Cryspen.

Overview

In this document, we describe the proof effort around the Rust implementation of ML-KEM in libcrux, which developed as part of the FOSPQC effort. We begin with a brief overview of the verification workflow and code structure and then detail various proof components.

Verification Workflow

The high-level implementation and verification workflow we follow is depicted below.

1. We first write implementations in safe, no_std, Rust and feed it to the hax toolchain, which type-checks the source code (in Rust) and checks it for constructs we do not support (like `unsafe`). For ML-KEM, the Rust code lives here.

2. The hax toolchain also uses the Rust typechecker to check whether the source code is "secret independent": does it branch on secret values, does it perform unsafe

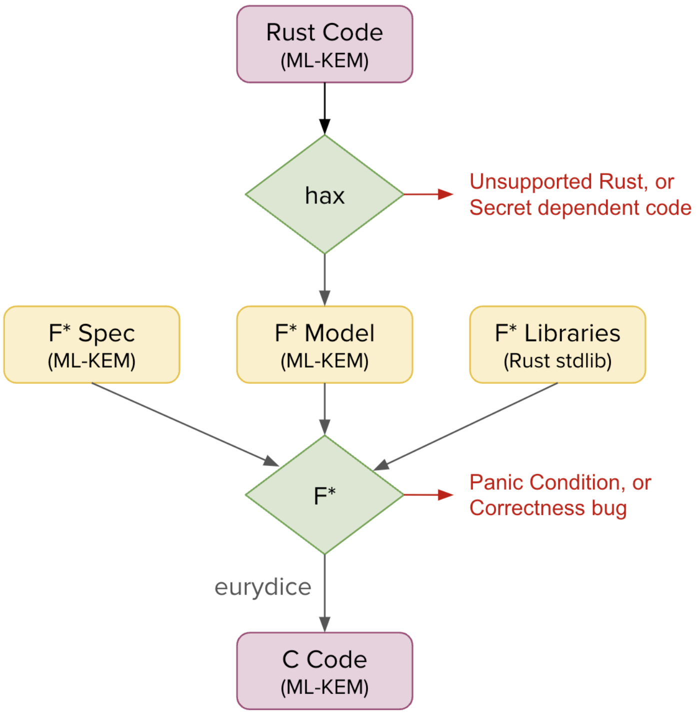
operations like divisions on secret values, does it access memory using secret indices etc. Enforcing secret independence at the Rust level ameliorates an important class of timing side channels, but does not guarantee that the compiled binary will not be vulnerable to timing, power, fault, or speculative side-channel attacks. To get more thorough side-channel guarantees, we recommend using other side-channel detection tools at the binary level.

3. Once these checks pass, hax is able to compile the Rust code into purely functional code in F* that closely reflects the Rust source code. For any Rust libraries or external crates used by the Rust code, we need to provide models in F*. The hax toolchain comes equipped with a set of F* libraries that model many commonly-used components of the Rust core library. For other libraries used in the Rust code, we need to write new F* code that models the behavior of these libraries.

4. Independently, we write a high-level mathematical specification of the cryptographic algorithm by hand in F*. This specification can be made executable and tested like any other implementation of the algorithm, but it ignores low-level optimizations and tries to be as succinct as possible while closely following the published standard. For ML-KEM, the F* spec is in this folder:

5. We then use F* to prove that all the Rust code is panic-free, that is, it does not access arrays out-of-bounds, and it does not overflow or underflow integers, etc. Otherwise, executing the code may result in a run-time panic in Rust (or worse, a memory corruption in C). To prove panic-freedom, we typically need to annotate the Rust code with pre-conditions and invariants on the lengths of slices, the ranges of values of various integers, etc.

6. Finally, we use F* to formally verify that the F* model generated from our Rust code meets the hand-written F* specification, under the assumption that our F* libraries accurately model the Rust core library and other external dependencies. If the proof fails, it either indicates a bug in the source code, or that we may need to provide annotations and lemmas to help F* prove the necessary property. At this stage, we are essentially constructing an F* proof by hand, with the aid of the Z3 SMT solver to discharge simple properties.

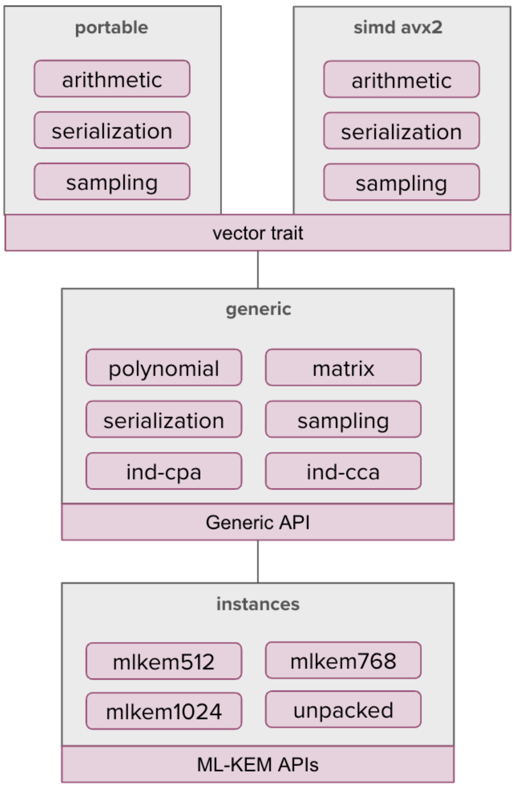


7. Once the code has been verified to our satisfaction, we can use the eurydice tool to compile the Rust code to a set

of C files. The C code closely follows the structure of the Rust, except that it monomorphizes and specializes all the generic Rust functions, flattens all the namespaces, and links the Rust library constructions to the standard C library. For ML-KEM, the generated C code is in this folder.

We follow the above process for the ML-KEM implementation and will do the same for ML-DSA. In the rest of this document, we will highlight the main tasks in each step.

Code Structure



To aid code-sharing and verification, we structure our Rust code for ML-KEM into multiple layers of code connected via abstract APIs as depicted below. The Rust code further provides an API that multiplexes between the different implementations, depending on the platform it is running on. However, this code is not extracted to C because most integration targets provide their own way of performing feature detection.

ML-KEM operates on polynomials of size 256, where each coefficient is a 12-bit integer, and all arithmetic is modulo the prime 3329. In our code we represent coefficients using the `i16` type of signed 16-bit integers, and we implement efficient algorithms for modular arithmetic over these coefficients.

- We first implement the core arithmetic, serialization, and sampling operations of ML-KEM on blocks of 16 coefficients (we choose 16 since they fit in a single 256-bit vector on AVX2 platforms). We build two implementations of these modules: one that is portable (i.e. relies only on standard 32-bit operations) and the other that relies on the AVX2 instructions exposed by the Rust intrinsics in `core::arch::x86_64`.
  We have also built an implementation of these modules for ARM64 platforms using the Neon instruction set, but that code is outside the scope of this document. Both implementations (portable and avx2) meet the same API exposed as a `Vector` trait that exposes the abstract operations but hides the internal implementation details.
- Using the abstract vector API, we then build the rest of ML-KEM in a generic way, without depending on the concrete representation of vectors. This generic layer includes code for polynomial and matrix operations, full serialization and sampling of keys, and the IND-CPA PKE and IND-CCA KEM APIs defined by the ML-KEM standard.
- Finally, we define instances of the generic code for all three variants of ML-KEM (512, 768, 1024) and build user-facing APIs that include useful features such as multiplexing based on platform detection and optional APIs for unpacked keys etc. (not currently extracted to C).

In terms of verification, this structure means that any proofs we do for the generic layer hold for all ML-KEM variants and all platforms, they do not have to be redone for AVX2/Neon/Portable etc. This is in contrast to many implementations of ML-KEM where all the code is different between the different platforms. In particular, we only need to be concerned with AVX2 instructions in the modules on the top right, and we can ignore SIMD parallelization everywhere else. We highlight the main verification effort needed in the top two layers and describe verification of the two vector trait implementations.

External Dependencies

The Rust code is written with no_std, i.e. such that it does not require the Rust standard library, to minimize dependencies and allow running it on embedded devices eventually.

The Rust code further depends on the hax library crate that provides utilities to annotate the code. Note however that this dependency is only enabled when using the separate hax configuration and is not used during regular builds.

Beyond this, the Rust code depends on four crates

- rand_core
- libcrux-platform
- libcrux-sha3
- libcrux-intrinsics

Only rand_core is an external dependency. It is used for the randomized APIs, which are not extracted to C.

The platform and intrinsics crates provide hardware abstraction for CPU feature detection and AVX2/NEON intrinsics. This code is not extracted to C. But the proofs rely on them being implemented correctly, especially the intrinsics crate (which is a thin, safe wrapper around the Rust intrinsics).

The libcrux-sha3 crate was written by us and provides implementations of all the SHA-3 functions that are portable, as well as optimized for AVX2. This code is also extracted to C and required by the ML-KEM code to run. We wrote this crate because the ML-KEM performance is heavily dependent on the performance of Sha3, and in particular special APIs that perform multiple Sha3 operations in parallel, and these operations are not provided by standard SHA-3 implementations. This code is currently not verified.

F* Dependencies

The F* specifications for ML-KEM and some utilities can be found in `proofs/fstar/spec`. Other F* libraries we rely on are:

- The F* standard library for sequences, lists, mathematical lemmas etc.
- The HACL* library for models of machine integers, loop combinators, and specifications of SHA-3
- The hax F* library for models of the Rust standard library

Verification

Verifying Panic Freedom

For all the code we write in Rust, the first property we prove is panic-freedom. In ML-KEM, the sizes of all inputs, outputs, keys, ciphertexts, and internal arrays are fully determined by the variant. For example:

- In ML-KEM 768, the algorithm manipulates 3x3 matrices, using public keys of size 1184 bytes, secret keys of 2400 bytes, to produce ciphertexts of 1088 bytes. Internally, it compresses ciphertext components at 10 and 4 bits per coefficient.
- In ML-KEM 1024, the algorithm manipulates 4x4 matrices, using public keys of size 1568 bytes, secret keys of 3168 bytes, to produce ciphertexts of 1568 bytes. Internally, it compresses ciphertext components at 11 and 5 bits per coefficient.

In our code, we write generic Rust code that works for both these versions by providing all these lengths as "const generic" arguments that are eliminated at compile time. The main benefit is that we only need to write a single implementation for all three variants of ML-KEM. The drawback is that the code now has a large number of constant arguments, which are left implicit in the standard. For example, the type for IND-CPA encryption is as follows:

```rust
pub(crate) fn encrypt<
   const K: usize,
   const CIPHERTEXT_SIZE: usize,
   const T_AS_NTT_ENCODED_SIZE: usize,
   const C1_LEN: usize,
   const C2_LEN: usize,
   const U_COMPRESSION_FACTOR: usize,
   const V_COMPRESSION_FACTOR: usize,
   const BLOCK_LEN: usize,
   const ETA1: usize,
   const ETA1_RANDOMNESS_SIZE: usize,
   const ETA2: usize,
   const ETA2_RANDOMNESS_SIZE: usize,
   Vector: Operations,
   Hasher: Hash<K>,
>(
   public_key: &[u8],
   message: [u8; SHARED_SECRET_SIZE],
   randomness: &[u8],
) -> [u8; CIPHERTEXT_SIZE] {...}
```

However, once all functions have been written with these constant arguments, the other key benefit is that the sizes of all internal arrays in the ML-KEM code are statically known at compile time and hence we can use native (fixed-size) arrays instead of relying on vectors (heap allocations). Indeed, note that the size of the ciphertext above is `[u8; CIPHERTEXT_SIZE]`.

However, even when we use fixed-size arrays, the Rust typechecker cannot prove that we are accessing arrays within bounds (array indexing in Rust may always panic). Furthermore, when we pass arrays as slices the type loses information about its length. For example, note that the public key is passed in as a slice of type `&[u8]` above, without any length information, even though we know it must contain an array of a specific size for each variant. Consequently, in the code for `encrypt` it is quite possible that we read the public key or message at the wrong index and this would result in a panic at runtime. When the code is eventually compiled to C, this would cause a memory error.

To prevent such panics, we annotate every function in the ML-KEM implementation with concrete length pre-conditions. For example, the precondition for the encrypt function is:

```
#[hax_lib::requires(fstar!("Spec.MLKEM.is_rank $K /\\
   $ETA1 = Spec.MLKEM.v_ETA1 $K /\\
   $ETA1_RANDOMNESS_SIZE = Spec.MLKEM.v_ETA1_RANDOMNESS_SIZE $K /\\
   $ETA2 = Spec.MLKEM.v_ETA2 $K /\\
   $BLOCK_LEN == Spec.MLKEM.v_C1_BLOCK_SIZE $K /\\
   $ETA2_RANDOMNESS_SIZE = Spec.MLKEM.v_ETA2_RANDOMNESS_SIZE $K /\\
   $U_COMPRESSION_FACTOR == Spec.MLKEM.v_VECTOR_U_COMPRESSION_FACTOR $K /\\
   $V_COMPRESSION_FACTOR == Spec.MLKEM.v_VECTOR_V_COMPRESSION_FACTOR $K /\\
   $CIPHERTEXT_SIZE == Spec.MLKEM.v_CPA_CIPHERTEXT_SIZE $K /\\
   $T_AS_NTT_ENCODED_SIZE == Spec.MLKEM.v_T_AS_NTT_ENCODED_SIZE $K /\\
   $C1_LEN == Spec.MLKEM.v_C1_SIZE $K /\\
   $C2_LEN == Spec.MLKEM.v_C2_SIZE $K /\\
   length $public_key == Spec.MLKEM.v_CPA_PUBLIC_KEY_SIZE $K /\\
   length $randomness == Spec.MLKEM.v_SHARED_SECRET_SIZE /\\
"))]
```

This precondition links each const generic argument of the function to the corresponding value in the ML-KEM standard (as encoded in the F* specification) and then provides length preconditions for the `public_key` and `randomness` arguments, since they are slices.

Given these pre-conditions, F* can prove that the code never goes out of bounds on any of the arrays used in IND-CPA encryption. Similarly, for arithmetic functions, we may need to add preconditions about the sizes of the inputs to prevent arithmetic overflow.

In addition to adding such preconditions throughout the code, we sometimes need to help F* prove panic freedom in code that uses loops. In such cases, we provide loop invariants that are sufficient for F* to prove the necessary properties. For example, the `compress_then_serialize_u` function used in IND-CCA encryption uses the following loop, where each iteration modifies the array `out`:

```
for (i, re) in input.into_iter().enumerate() {
    hax_lib::loop_invariant!(|i: usize| out.len() == OUT_LEN);
    out[i * (OUT_LEN / K)..(i + 1) * (OUT_LEN / K)].copy_from_slice(
            &compress_then_serialize_ring_element_u::<COMPRESSION_FACTOR,  BLOCK_LEN,
Vector>(&re),
        );
}
```

Here, we add a simple loop invariant that states that the length of the mutable output slice `out` does not change in the body of the loop. This is obviously true from looking at the Rust code, but is needed for F* to be able to prove that we are not writing into `out` outside its bounds.

Verifying Vector Arithmetic

At the heart of the ML-KEM implementation are a few algorithms for polynomial arithmetic using the Number Theoretic Transform (NTT). Given a polynomial represented as 256 field elements (modulo 3329), the standard defines an algorithm for converting the polynomial to NTT form, another algorithm for the inverse NTT, and an algorithm for multiplying two polynomials in NTT form. All three algorithms require modular arithmetic over the field of integers modulo 3329.

Since the modulus operation can become a performance bottleneck, typical implementations, such as ours, use Montgomery arithmetic over field elements, and rely on barrett reduction for the final reduction modulo 3329. In all these computations, we need to be particularly careful not to overflow the limit of our machine integer representation. In our code, we use `i16` as the base representation and use signed Montgomery and Barrett reductions, which are a little different from the classic forms of these algorithms over unsigned integers.

In our SIMD vectorized code, we define Montgomery multiplication between a vector of 16 field elements (implemented as a 256-bit value) and a constant, where both are already in Montgomery form as follows:

```rust
pub(crate) fn montgomery_multiply_by_constant(vector: Vec256, constant: i16) -> Vec256 {
   let constant = mm256_set1_epi16(constant);
   let value_low = mm256_mullo_epi16(vector, constant);
   let k = mm256_mullo_epi16(
       value_low,
       mm256_set1_epi16(INVERSE_OF_MODULUS_MOD_MONTGOMERY_R as i16),
   );
   let k_times_modulus = mm256_mulhi_epi16(k, mm256_set1_epi16(FIELD_MODULUS));
   let value_high = mm256_mulhi_epi16(vector, constant);
   mm256_sub_epi16(value_high, k_times_modulus)
}
```

In this code, we perform a sequence of arithmetic operations that will be familiar to anyone who has seen Montgomery multiplication before, except that this code performs these operations on all 16 values in parallel and tries to use the most efficient instructions available on AVX2.

To prove this code correct, we proceed in two steps. We first show that this code implements the same function `mont_red` on all 16 values in the vector. We then prove that `mont_red` computes the right value. In particular, we prove that as long as the absolute value of the constant (|`constant`|) is less than 3329, the absolute value of the output of `mont_red` will be less than 3329 + 1665, and furthermore that this output is equivalent to the input multiplied by the Montgomery inverse (modulo 3329).

Structuring the proof in this way also allows us to reuse the same proof argument for the portable version of our code. We do a similar proof for Barrett reduction. These two are probably the most mathematically involved pieces of code in our implementation. The rest of the implementation involves point-wise arithmetic on arrays, as well as matrix and vector multiplication, which are relatively straightforward to verify using loop invariants.

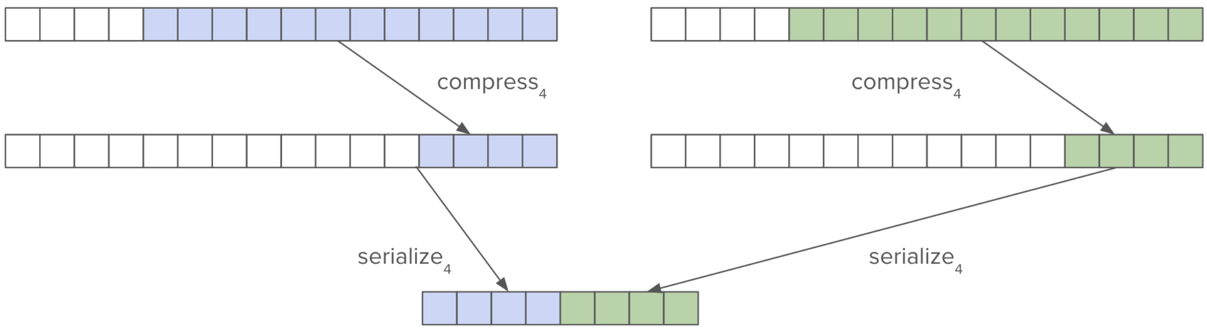
In principle, NTT arithmetic can be a subtle element of the ML-KEM construction. However, we do not involve ourselves in the mathematical meaning of NTT. Instead, we directly encode the algorithms described in the standard and prove that our Rust code implements these algorithms. In other words, our proof shows that the code meets the standard, but we leave reasoning about the correctness or security of the standard to other works. In our proofs, the main challenge in proving the NTT functions correct is to a) carefully track the bounds of coefficients as they grow through the NTT layers and ensure that they do not overflow `i16`, b) prove that the portable implementation correctly implements the algorithm, and c) prove that the optimized AVX2 implementation correctly implements the algorithm.

Verifying Compression and Serialization

Although much attention in lattice-based crypto goes towards the underlying mathematics, a key aspect of the ML-KEM standard is the compression and encoding functions that specify the wire-format of the keys and ciphertexts, allowing different implementations to interoperate. These encoding functions are easy to describe but are surprisingly hard to implement efficiently, and each ML-KEM implementation ends up defining serialization and deserialization functions for 1, 4, 5, 10, 11, and 12 bits per coefficient. This means 10 functions in portable code and 10 more functions in AVX2. The effort to implement and reason about these functions is disproportionately high as compared to the rest of the ML-KEM implementation.

It is also important to verify these functions, since in cryptographic constructions, deserialization (and re-serialization) is often performed on adversary-supplied values and can trigger memory errors and side-channel leaks. Indeed, one of the compression steps during decryption was at the root cause of the timing issue, aka KyberSlash, in a number of Kyber implementations (a bug initially found by us as part of our formal verification effort).

The general structure of the (de-)serialization code in ML-KEM is quite simple. We first compress a polynomial so that each coefficient has less than `d` bits. We then pack this sequence of `d` bits into a byte array. Of course, if `d` were 8, the packing would be trivial, but in practice, `d` is one of 1, 4, 5, 10, 11, or 12. For example, the encoding steps for `d=4` are depicted below. The decoding steps work in reverse.



When d=4, every coefficient yields 4 bits, and hence every pair of coefficients can be serialized into one byte. So a full polynomial is encoded as 128 bytes and so on. When d=10, we can process 4 coefficients at a time to produce 5 bytes. In our AVX2 implementation, we always process 16 coefficients at a time, and so we produce 2*d bytes for each vector, so between 2 and 24 bytes.

In the flow above, the compress steps are pointwise and relatively easy to implement. However, the (de)serialization code can get quite tedious even though in the end they are just moving bits from one location to another without any interesting mathematical operations. For example, the portable code for serialize 10 uses the following function four times:

```
pub(crate) fn serialize_10_int(v: &[i16]) -> (u8, u8, u8, u8, u8) {
   let r0 = (v[0] & 0xFF) as u8;
   let r1 = ((v[1] & 0x3F) as u8) << 2 | ((v[0] >> 8) & 0x03) as u8;
   let r2 = ((v[2] & 0x0F) as u8) << 4 | ((v[1] >> 6) & 0x0F) as u8;
   let r3 = ((v[3] & 0x03) as u8) << 6 | ((v[2] >> 4) & 0x3F) as u8;
   let r4 = ((v[3] >> 2) & 0xFF) as u8;
   (r0, r1, r2, r3, r4)
}
```

Rather than trying to prove each of these functions by hand (and there are many such functions in ML-KEM), we chose to automate the proofs using F* tactics. We write the specifications of these functions in terms of individual bits. For example, we say that the i-th bit of the output of $\mathtt{serialize}_{10}$ must be the same as the j-th bit of the k-th coefficient of the input. Then we seek to prove this property automatically for each bit of the output. Our tactic computes an expression for the i-th bit of the output, simplifies this expression using the F* normalizer, and then passes the expression to Z3, to prove that it matches the specification for this bit. This process is repeated for each bit of the output (from 0 to 2 * d * 8 - 1). When the number of bits in the output is large, the proof can take a long time (tens of seconds per bit) but the time taken is linear in the number of bits.

We wrote one set of tactics to do the above for the portable serialization and deserialization code, and another set that works on AVX2 code. Using these tactics, we were able to prove our serialization code correct without having to manually reason about their correctness.

We note three things about this approach. First, by using tactics, we simplify the proof effort in a sound way, in that F* verifies the proof generated by the tactic. Second, these proofs show where using tools like Z3 can be useful, since SMT solvers are particularly good at discharging goals about large expressions that only use boolean operations (and some of our expressions are very large). Third, while the effort of developing these tactics was non-trivial, it pays off over time, since we can now use the same tactics to verify serialization code for all ML-KEM variants and for other similar constructions like ML-DSA. In essence, we chose to use a lot of machine time for these proofs but we believe that this is much better than using human effort for such tedious proofs.

Verifying the Generic ML-KEM Code

The main intellectual effort of verifying ML-KEM was in the vectorized arithmetic code and in developing tactics for handling serialization. The rest of the verification amounts to making sure that the generic Rust code (say for IND-CPA) correctly follows the algorithm in the standard. Since both the implementation and standard are written in a sequential imperative style, this is relatively straightforward. The main proof effort is to annotate the generic code with the right pre- and post-conditions, ensure that it meets all the pre-conditions of the functions it depends on, and that the post-condition implies the spec.

The sampling algorithms in ML-KEM have one feature that is tricky for verification. The sampling of matrix A uses SHAKE128 as an XOF in a while loop and may (in principle) never terminate, even though the probability that this loop is executed more than (say) 5 times is provably negligible. In F*, we cannot define a specification that does not terminate, and so we impose an abstract artificial limit in the spec after which the sampling always terminates with an “insufficient randomness” flag. We then prove that the Rust code for sampling provides the right result as long as the F* spec also terminates with a valid result.

How do I use it?

All the proof work is currently being merged to the dev branch.

Rust

The Rust code lives inside the libcrux repository, in the libcrux-ml-kem crate. The Rust documentation shows how to use it. Tests can be run with `cargo test`, and benchmarks with `cargo bench` (note that `--features pre-verification` is required on main until the dev branch with the proofs is merged into main).

Verification

The repository contains the extracted F* code. CI ensures that the extraction is always in sync with the Rust code.

F* Extraction

The verification artifacts live in the `proofs/fstar/extraction` directory within the libcrux-ml-kem crate. The F* code in this directory is extracted from Rust with hax by running `./hax.py extract` in the libcrux-ml-kem directory. In order to run the extraction, a local installation of hax is required (see the hax readme for details on how to install hax).

F* Verification

To run the F* verification, `./hax.py prove` can be used, which runs the Makefile in the `proofs/fstar/extraction` directory. To get started with F* we refer to the official F* website.

C Extraction

To extract the C code from the Rust code additional tools are necessary. We refer to the C-CODE.md file in the libcrux repository for the setup of the toolchain for C extraction.

The C code is then extracted by running `./boring.sh` in the libcrux-ml-kem crate

Novelty and Challenges

Our proof methodology is inspired by the HACL* project which also uses F* to verify cryptographic implementations. However, there are several key novelties in our approach, which may be of independent interest to researchers and engineers.

Verifying a Large Idiomatic Rust Crate

Unlike HACL*, which is written in a carefully curated subset of F* and therefore targets code that is explicitly written for ease of verification, in this project we targeted Rust code written by crypto engineers without much exposure to formal verification. The only concession we made was to ensure that the code was acceptable to the HAX frontend.

Once the code was written, the proof engineering team went in and added annotations directly in the Rust code, but the goal was to do this without changing the structure or functionality of the Rust code. (There were very few exceptions where the code needed to be restructured for proof-friendliness).

This discipline of trying to verify idiomatic Rust code added additional burdens on our proof tools. In particular, we needed to handle Rust traits, since they are commonly used in Rust code, and we needed to add and improve mechanisms for adding proof annotations (pre- and post-conditions, invariants) directly in Rust code. This effort inspired us to improve our tools significantly.

The second impact of using this style was that the verification target artifact was much larger. The ML-KEM code in Rust was about 9000 lines in 45 files, whereas the generated F* is about 20000 lines in 134 files for most of which we needed to do proofs. It is likely that an equivalent implementation written directly in F* could have been much more succinct.

On the whole, we believe ours is one of the largest proof efforts for crypto code directly written in Rust.

Verifying Vectorized Rust Generically

At a high-level, our code structure and proofs for SIMD follow the ideas in HACLxN, a SIMD extension of HACL*. The idea is to write as much of the implementation in a generic way as possible, only defining platform specific implementations for low-level arithmetic and other operations that need to be optimized.

As a result, about half our Rust code was generic and shared between the portable and AVX2 implementations, and this code only needed to be verified once. This is in contrast to other implementations of ML-KEM, where the code for different platforms essentially copies over the high-level algorithm code.

This approach means that the pre- and post-conditions for the low-level SIMD-optimized functions must be the same for portable, AVX2, and even Neon. This imposes a discipline on our code, but allows us to do our proofs more modularly.

Automating Proofs for Bit-level Serialization

Serialization and deserialization of messages, ciphertexts, and keys forms a small and seemingly straightforward part of the ML-KEM standard but actually ends up consuming a disproportionate amount of programming and proof time. We would estimate that a third or even half of the AVX2 implementation time was spent in defining optimized serialization and deserialization functions.

The specifications of these functions is straightward, they are essentially packing and unpacking bits, moving them from one representation to another. Proving these functions manually is very tedious. Consequently, we developed automated tactics to verify these kinds of functions in F*. Although developing the tactic took some time, and running it on large bitvectors also takes processor time, using the tactic allowed us to focus on the interesting parts of the proofs.

Proof Challenges

Overall, we can divide the proofs for various modules into four types:

- Mathematically Interesting: field, polynomial, matrix arithmetic (NTT) *(5k lines)*
- Low-level and Annoying: serialization and deserialization *(5k lines)*
- Straightforward and Boring: algorithms for sampling, IND-CPA, IND-CCA *(4k lines)*
- High-level APIs: ML-KEM 512/768/1024, multiplexing, unpacked (4k)
- Performance sensitive code: SHA-3, unpacked API, NTT, memcpy

For the interesting proofs, we mainly developed manual proofs in F*, which consumes time but is effective.
The boring proofs are straightforward and do not require manual arguments, but sometimes require a large amount of memory to verify (unless we split each algorithm into smaller components). They are also quite time-consuming just because of the size of the code. For the annoying proofs, we used automation wherever possible.

It is worth noting, that of all the code we consider, the code that ends up being the most important for performance is SHA-3, and is outside the scope of our verification. Every other component, including NTT, is completely dominated by the time taken by SHAKE-128 and other SHA-3 functions.

After SHA-3, the biggest performance impact is in using the unpacked API. Again, this is something not covered by our proofs yet, but we plan to verify the code for this API. Next is NTT, and then the most expensive operations were redundant memcpys, and we eliminated these using mutable caller-allocated arrays whenever they affected performance.