Jazzline: Composable CryptoLine functional correctness proofs for Jasmin programs

Anonymous Author(s)

Abstract

Jasmin is a programming language designed for the implementation of high-speed and high-assurance cryptography. Functional correctness proofs of Jasmin programs are typically carried out deductively in the EasyCrypt theorem prover. This allows generality, modularity and composable reasoning, but does not scale well for low-level architecture-specific routines. CryptoLine offers a semi-automatic approach to formally verify algebraically-rich low-level cryptographic routines. CryptoLine proofs are self-contained: they are not integrated into higher-level formal verification developments. This paper considers the problem of soundly using CryptoLine to discharge subgoals in functional correctness proofs for complex Jasmin programs. To this end, we extend Jasmin with annotations and provide an automatic translation into a CryptoLine model, carrying out the most complex transformations in the certified core of the Jasmin compiler itself. We also formalize and implement the automatic extraction of the semantics of a CryptoLine proof to EasyCrypt. Our motivating use-case is the X-Wing hybrid KEM, for which we present the first formally verified implementation.

CCS Concepts

• **Security and privacy** → **Logic and verification**; **Cryptography**.

Keywords

Computer-Aided Cryptography, Formal Verification, EasyCrypt, Jasmin, CryptoLine

ACM Reference Format:
Anonymous Author(s). 2025. Jazzline: Composable CryptoLine functional correctness proofs for Jasmin programs. In . ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Cryptographic libraries are a cornerstone of computer security. As such, it is essential that these libraries are correctly implemented. Unfortunately, guaranteeing functional correctness of cryptographic implementations remains extremely challenging. Therefore, there is a growing emphasis on using program verification for proving correctness of these implementations [5]. At one end of the spectrum, automated verification tools such as CryptoLine [10, 14] have been used to verify challenging and highly optimized subroutines, including multi-precision arithmetic, elliptic curve cryptography, and core components in lattice-based cryptography such as the Number Theoretic Transform. However, these approaches are limited to specific classes of programs—informally, straight-line code with statically resolved memory accesses, and are not compositional. At the other end of the spectrum, interactive verification tools such as EasyCrypt have been used to verify full algorithms, including high-speed implementation of the SHA-3 standard [2] and the recent FIPS-203 ML-KEM standard [1, 3]. However, these approaches are labour intensive and do not scale well for low-level subroutines. The goal of this work is to integrate both approaches in a single verification framework.

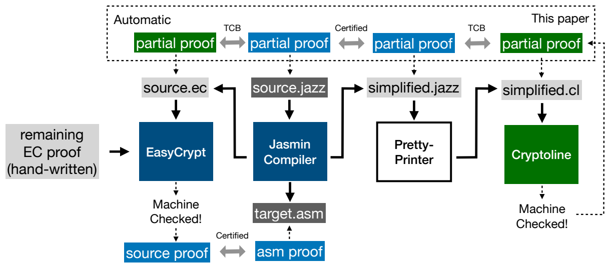

Figure 1: Diagram of the extended tool-chain.

Contributions. This paper develops sound foundations and tooling for combining deductive and automated program verification, namely EasyCrypt and CryptoLine, and shows how the resulting framework helps verifying cryptographic implementations written in the Jasmin language. Our approach is based on the following steps and illustrated by a diagram in Figure 1. The traditional proof development architecture for a Jasmin source program is shown on the left of the diagram. A Jasmin source file is compiled to assembly by the certified Jasmin compiler, which means that functional correctness established at the source will also hold at the assembly level. In parallel, a model of the source program is extracted to the EasyCrypt theorem prover, where the functional correctness proof for the source code is machine-checked.

In this paper we extend the Jasmin language and compiler to allow integration of CryptoLine in the tool-chain:

- We extend the Jasmin programming language with assume and assert statements via annotations in the source code. These statements allow users to state functional correctness contracts using an extension of the Jasmin language of expressions, which is general enough to write the annotations required for a CryptoLine-based correctness proof. Furthermore, our annotation language is higher-level than that used typically in CryptoLine proofs, as it includes big operators for summation and universal quantification, and so the resulting contracts are more convenient to work with in high-level proofs.

- We implement new verified compilation steps in the Jasmin compiler (these are specific to the new proof methodology we introduce here) that perform a Jasmin-to-Jasmin transformation that we call simplification. The resulting simplified Jasmin programs are equivalent to the source, but they are stripped of all high-level constructs, including branching, loops, arrays, etc., and they are also in static single-assignment form. However, we retain function calls to allow for a modular proof relying on CryptoLine. Furthermore, and crucially for this work, the compiler is proved to preserve the semantics of annotations.
- We state and prove a soundness meta-theorem that justifies the methodology illustrated in Figure 1. The annotated Jasmin source program is simplified and then pretty-printed to CryptoLine. If CryptoLine accepts the program, then this yields a partial correctness result. Here the proof is partial for two potential reasons: 1) the possible use of assume statements in the proof and 2) the fact that verification in CryptoLine is done independently for each function in the call tree. Our meta-theorem then shows that the certified simplification of Jasmin programs permit lifting the CryptoLine proof back to the source Jasmin program.
- We extend the extraction to EasyCrypt, so that it includes the partial correctness proved by CryptoLine as axioms. Again, the way in which the extraction to EasyCrypt is performed is justified by our meta-theorem. As a result, the effort of constructing a full EasyCrypt proof for the source Jasmin program is much reduced. In particular, we can synthethise the EasyCrypt proof that justifies our modular verification in CryptoLine of each function in the call tree. This means that the only non-trivial goals that remain to prove in EasyCrypt concern justifying the assumptions made when carrying out the CryptoLine proof.

The benefits of our approach are numerous, and they affect both the users of Jasmin and the users of CryptoLine:

- Functional correctness proofs for complex Jasmin programs can now be factored into high-level compositional reasoning—which is natural in EasyCrypt and typically carries over across implementations for different architectures—and automatic proofs for low-level functions in which CryptoLine excels.
- This level of automation still retains the strong end-to-end guarantees of EasyCrypt proofs: the program has a well defined semantics for all valid inputs and is correct with the input/output behavior given by the pre- and post-condition. We achieve this by making sure that all assumptions made in CryptoLine proofs, which are the responsibility of the user, need to be justified.
- The way in which we carry out CryptoLine proofs permits composing them using EasyCrypt. This is necessary for our end-goal of integrating the CryptoLine results in more complex proofs, but it also has implications for CryptoLine users. For example, one can use the approach proposed in this paper to carry out different proofs in CryptoLine for the same function, corresponding to different cases/pre-conditions, and then use EasyCrypt to combine these proofs into a single one that aggregates the various cases. Similarly, EasyCrypt can be used to justify the modular reasoning about various functions in a call tree.
- Effort can be shared and reused across different tools and teams.

We illustrate our approach with two examples. The first and main motivating example is X-Wing, where we use previous CryptoLine proofs of X25519 subroutines and compose them with previous EasyCrypt proofs of ML-KEM and SHA3. The second example is ML-KEM NTT, where we illustrate modular reasoning where parts of the call tree are justified in CryptoLine and other parts in EasyCrypt. Furthermore, we also show how we can reuse existing CryptoLine proofs for the NTT by bridging in EasyCrypt the algebraic description of the NTT (based on the Chinese-Remainder Theorem) used in CryptoLine contracts to the DFT-based description of the NTT used in the EasyCrypt proof of correctness wrt the ML-KEM standard.

Related work. There is a large body of work that uses program verification for proving correctness of cryptographic code [5]. A majority of this work, including [4, 15, 17] and prior work on Jasmin, follows the classic approach of deductive verification. The design of CryptoLine [14] features automation over scalability and compositionality, so in this work we explore the possibility of combining two verification frameworks in a sound way. An alternative approach would have been to extend EasyCrypt directly with the ability to carry out CryptoLine-style proofs, which is the spirit of the work done in [16] for the Dafny framework. Another traditional approach is to generate correct programs from specifications. This is the approach followed by FIAT Crypto [9]; recently the approach has been extended with mechanisms to synthesize correct and optimized implementations [11]. This approach is appealing but is not applicable to the complex implementations that we consider.

Structure of the paper. Section 2 provides further motivation via an example. Sections 3 to 5 justify the theoretical soundness of our approach and the extensions to the Jasmin compiler. Sections 6 and 7 describe our implementation of the interaction with CryptoLine and EasyCrypt, respectively. Finally, Section 8 describes the examples and Section 9 adds some concluding remarks.

Access to development. The development is available from the following anonymous link.

```
Algorithm expandDecapsulationKey($sk$):
    $(\mathsf{coins}_M, sk_X) \leftarrow \mathsf{SHAKE256}(sk, 96)$
    $(pk_M, sk_M) \leftarrow \mathsf{ML\text{-}KEM\text{-}768.KeyGen}(\,;\mathsf{coins}_M)$
    $pk_X \leftarrow \mathsf{X25519}(sk_X, \mathsf{X25519\_BASE})$
    Return $(sk_M, sk_X, pk_M, pk_X)$

Algorithm KeyGen( ):
    $sk \xleftarrow{\$} \{0,1\}^{256}$
    $(sk_M, sk_X, pk_M, pk_X) \leftarrow \mathsf{expandDecapsulationKey}(sk)$
    Return $((pk_M, pk_X), sk)$

Algorithm Encapsulate($(pk_M, pk_X)$):
    $ek_X \xleftarrow{\$} \{0,1\}^{256}$
    $cph_X \leftarrow \mathsf{X25519}(ek_X, \mathsf{X25519\_BASE})$
    $shk_X \leftarrow \mathsf{X25519}(ek_X, pk_X)$
    $(shk_M, cph_M) \leftarrow \mathsf{ML\text{-}KEM\text{-}768.Encap}(pk_M)$
    $shk \leftarrow \mathsf{SHA3\text{-}256}(shk_M, shk_X, cph_X, pk_X, \mathsf{label}_X)$
    Return $(shk, (cph_M, cph_X))$

Algorithm Decapsulate($sk, (cph_M, cph_X)$):
    $(sk_M, sk_X, pk_M, pk_X) \leftarrow \mathsf{expandDecapsulationKey}(sk)$
    $shk_M \leftarrow \mathsf{ML\text{-}KEM\text{-}768.Decap}(cph_M, sk_M)$
    $shk_X \leftarrow \mathsf{X25519}(sk_X, cph_X)$
    $shk \leftarrow \mathsf{SHA3\text{-}256}(shk_M, shk_X, cph_X, pk_X, \mathsf{label}_X)$
    Return $shk$
```

Figure 2: The X-Wing Hybrid KEM

2 Motivating Example

Introducing X-Wing. To motivate our approach we will use X-Wing [6, 8], a recently proposed hybrid Key Encapsulation Mechanism (KEM). X-Wing uses ML-KEM as a blackbox. ML-KEM was formerly known as Kyber, and it was recently standardized by NIST as FIPS-203 [13] (Module-Lattice-Based Key-Encapsulation Mechanism). The goal of the X-Wing design is to provide a fallback to the security of ML-KEM, and guarantee at least classical security in the unlikely scenario that a weakness is uncovered in the post-quantum standard. To this end, X-Wing runs an elliptic-curve-based KEM-like construction in parallel with ML-KEM, so that the X-Wing shared key is derived using contributions from both the classical and the post-quantum constructions. This classical layer is based on the X25519 elliptic curve [12], and the overall design is optimized for performance: the final key is derived by passing to the final SHA-3 based key derivation function just enough information to guarantee security, while requiring only a single block absorption for the hash computation. The security analysis of X-Wing [6] shows that the construction is secure against post-quantum adversaries if ML-KEM is secure, and that it is secure against classical adversaries if a variant of the computational Diffie-Hellman problem holds in the X25519 group.

The X-Wing construction is given in pseudocode in Figure 2.[1] Our goal is to obtain a high-speed and high-assurance implementation of X-Wing with minimal effort, which means reusing as much as possible pre-existing implementations and functional correctness proofs. Looking at the design of the construction, there are three components that need consideration: ML-KEM-768, X25519, and two functions in the SHA3 family. Our starting point are the pre-existing formally verified implementations of ML-KEM [1, 3] and SHA3 [2]. These implementations are written in the Jasmin programming language for x86-64, and the corresponding functional correctness proofs are written in the EasyCrypt proof assistant [7]. However, the functional correctness of the existing Jasmin implementations of X25519 were not yet formally verified.

X25519 implementations. Implementations of X25519 are pervasive in cryptographic libraries, and they are typically structured in two layers: low-level leaf functions that perform multi-precision arithmetic modulo prime $p = 2^{255}$-19 and encoding/decoding routines, whereas a higher level function performs the relevant scalar multiplication operation in the group of elliptic curve points. The top level interface for this operation, shown as X25519$(\cdot,\cdot)$ in Figure 2, takes a 32-byte scalar u and a 32-byte representation of an elliptic curve point P, and returns the 32-byte representation of uP.

The challenging aspect of proving the correctness of an X25519 scalar multiplication operation resides in the multi-precision computations performed in the leaf functions. Indeed, in a tool such as EasyCrypt, once the leaf functions have been proved to perform the correct low-level computations, it is relatively straightforward to argue that the top-level function is calling the low-level functions in the correct sequence using Hoare logic. Furthermore, this high-level proof is essentially the same for any target architecture and multi-precision representation of finite-field elements. On the other hand, proving the correctness of the multi-precision arithmetic implementation requires significant effort in an interactive proof, which is hard to amortize across implementations for different target architectures and choices of finite field element representations.

[1]In this paper we follow the specification given in the Internet Draft [6, 8]

```
for i = 0 to 3 {                                   mulx   %r12,%rax,%rbx
  ( hi, lo )   = #MULX ( _38,  r[i] );             adcx   %rax,%r8
  of, h[i]     = #ADOX ( h[i], lo, of );           adox   %rbx,%r9
  cf, h[i+1]   = #ADCX ( h[i+1], hi, cf );         mulx   %r13,%rax,%rbx
}                                                  adcx   %rax,%r9
                                                   adox   %rbx,%r10
                                                   mulx   %r14,%rax,%rbx
                                                   adcx   %rax,%r10
                                                   adox   %rbx,%r11
( r[0], lo ) = #MULX ( _38, r[3] );                mulx   %r15,%rax,%r12
of, h[3]     = #ADOX ( h[3], lo, of );             adcx   %rax,%r11
cf, r[0]     = #ADCX ( r[0], z, cf );              adox   %rdi,%r12
of, r[0]     = #ADOX ( r[0], z, of );              adcx   %rdi,%r12
                                                   mov    0x10(%rsp),%rdi
_,_,_,_,_,lo = #IMULri ( r[0], 38 );               imul   %rdx,%r12
cf, h[0] += lo;                                    add    %r12,%r8
cf, h[1] += z + cf;                                adc    $0x0,%r9
cf, h[2] += z + cf;                                adc    $0x0,%r10
cf, h[3] += z + cf;                                adc    $0x0,%r11
_, z -= z - cf;                                    sbb    %rax,%rax
z &= 38;                                           and    $0x26,%rax
h[0] += z;                                         add    %rax,%r8
```

```
mull rbx rax rdx r12;
adcs carry r8 r8 rax carry;
adcs overflow r9 r9 rbx overflow;
mull rbx rax rdx r13;
adcs carry r9 r9 rax carry;
adcs overflow r10 r10 rbx overflow;
mull rbx rax rdx r14;
adcs carry r10 r10 rax carry;
adcs overflow r11 r11 rbx overflow;
mull r12 rax rdx r15;
adcs carry r11 r11 rax carry;
adcs overflow r12 r12 rdi overflow;
adcs carry r12 r12 rdi carry;
mull dontcare r12 rdx r12;
assert true && and [dontcare = 0@64,carry=0@1,overflow=0@1];
assume and [dontcare = 0,carry=0,overflow=0] && true;
adds carry r8 r8 r12;
adcs carry r9 r9 0x0@uint64 carry;
adcs carry r10 r10 0x0@uint64 carry;
adcs carry r11 r11 0x0@uint64 carry;
ghost carryo@bit: carryo = carry && carryo = carry;
sbbs carry rax rax rax carry;
assert true && carry = carryo;
assume carry = carryo && true;
mov overflow carry;
not zero@bit carry;
and rax@uint64 rax 0x26@uint64;
assert true && or [ and [carry=0@1, rax=0@64], and [carry=1@1, rax=0x26@64]];
assume rax = carry*0x26 && true;
adds carry r8 r8 rax;
assert true && carry=0@1;
assume carry=0 && true;
{ eqmod (limbs 64 [r8, r9, r10, r11]) (limbs 64 [a0, a1, a2, a3] * limbs 64 [a0, a1, a2, a3])
   ((2**255)-19)  &&  true }
```

Figure 3: Snippets of three implementations of the same X25519 basic subroutine, which computes the square of a 32-byte integer represented as 4x64-bit registers modulo $2^{255} - 19$. Top Left: Jasmin code. Top Right: openssl assembly. Bottom: CryptoLine proof script for openssl assembly.

Indeed, for such low level functions, searching for an automated proof methodology is well justified.

Pre-existing CryptoLine proof. In this work we consider CryptoLine [10, 14], a tool that allows precisely such an automated proof methodology. CryptoLine has been specifically designed for the verification of low-level implementations of mathematical constructs, including multi-precision arithmetic. Moreover, the CryptoLine set of examples includes proofs of the openssl implementation of X25519 finite field operations in x86-64 assembly, which are very close to the Jasmin implementation of the same functions. To illustrate this similarity, Figure 3 shows snippets from three versions of a routine that performs squaring modulo 2^{255}-19. The first one, on the left, is the Jasmin implementation we are interested in formally verifying. In the middle we show assembly code taken from the openssl implementation of the same routine.[2] Finally, on the right, we can see part of the CryptoLine proof script for the openssl assembly.[3] The snippets show part of the squaring routine that performs modular reduction from a 64-byte intermediate result to the 32-byte final result, i.e., it computes the modular reduction of $a^2 \pmod{2^{255} - 19}$ where $a \in [0; 2^{256})$ and is represented as four (packed) 64-bit words. It is clear that the Jasmin and openssl code are doing essentially the same thing, and this pattern repeats itself with minor differences in scheduling and instruction selection for the entire implementation of the X25519 leaf function. Our expectation is therefore that the CryptoLine proof actually applies to the Jasmin code, and our motivation in this paper is to allow reusing such proofs in a sound way.

Let us now take a closer look at the snippet of the CryptoLine proof script shown in Figure 3 (right). At the end of the code, one can see the post-condition. Like all predicates over CryptoLine program states, it has two parts $Q_A \mathbin{\&\&} Q_R$, where Q_A is called an algebraic predicate and Q_R is called a range predicate. Algebraic predicates refer to the (in this case, unsigned) integer interpretation of the values of program variables, whereas range predicates refer to program variables as fixed-length bit vectors.

In this example, Q_R is the trivial true post-condition, and the algebraic post-condition Q_A is the congruence $a^2 \equiv r \pmod{2^{255} - 19}$, where a is the integer represented by inputs to the function (a_0, a_1, a_2, a_3), and r is the integer represented by the function outputs $(r_8, r_9, r_{10}, r_{11})$. Here, all inputs and outputs are 64-bit values interpreted as unsigned integers, as the representation uses packed 64-bit limbs, s.t. for example $a := \sum_{i=0}^{3} 2^{64i} a_i$.

The CryptoLine proof consists of two parts: a model of the program, consisting of a sequence of CryptoLine instructions, and a set of annotations that introduce ghost variables and insert assert and assume statements.[4] The use of assert and assume statements in this proof is illustrative of an important point about CryptoLine proofs. Assert statements (in this case, of range expressions) are required by CryptoLine to hold in all relevant executions. On the other hand, assume statements can be used to constrain the set of executions for which the proof is required to hold (i.e., possibly beyond the restriction imposed by the pre-condition): an algebraic assumption such as those introduced in the current proof is taken as an additional equality that can be assumed to hold over the integers when the program is being checked for correctness. A closer look shows that algebraic assume statements in Figure 3 (bottom) are actually implied by the range assert statements inserted immediately before them, but this is not checked by the tool and is left to the user to confirm by inspection. This approach is extremely powerful, as it permits using the power of the SMT solver backends (used to discharge) range predicates to simplify the algebraic model (e.g., by incorporating an equation that sets a carry variable to 0) and can be critical to enable an automatic proof based on a Computer Algebra System (CAS).

```
param bool sub = false; param int MAX = 100; param int LEN = 2;
abstract predicate int u16i(u16);

fn source(reg ptr u16[LEN] a b) → reg ptr u16[LEN]
requires {\all (i \in 0:LEN) (0 <= a[i] && a[i] < MAX)}
requires {\all (i \in 0:LEN) (0 <= b[i] && b[i] <= a[i])}
ensures {\all (i \in 0:LEN) (0 <= u16i(result.0[i])  &&
                     u16i(result.0[i]) <= 2*MAX)}
{ for i = 0 to LEN { if sub { a[i] -= b[i]; } else   { a[i] += b[i]; } assert (0 <= a[i]); }
  return a; }
-----------------------------------------------------------------------------
fn simplified (reg u16 a_0_id_151, reg u16 a_1_id_152,
          reg u16 b_0_id_153, reg u16 b_1_id_154) → (reg u16, reg u16)
requires {(((((16u) 0) <=u a_0_id_151) && (a_0_id_151 <u ((16u) 100))) &&
          ((((16u) 0) <=u a_1_id_152) && (a_1_id_152 <u ((16u) 100))))}
requires {(((((16u) 0) <=u b_0_id_153) && (b_0_id_153 <=u a_0_id_151)) &&
          ((((16u) 0) <=u b_1_id_154) && (b_1_id_154 <=u a_1_id_152)))}
ensures {((((0 <= u16i(a_0_id_155)) && (u16i(a_0_id_155) <= 200)) &&
          ((0 <= u16i(a_1_id_156)) && (u16i(a_1_id_156) <= 200)))}
{reg u16 a_0_id_155; reg u16 a_1_id_156;
  (_, _, _, _, _,a_0_id_155) = #ADD_16(a_0_id_151, b_0_id_153);
  assert (((16u) 0) <=u a_0_id_155);
  (_, _, _, _, _,a_1_id_156) = #ADD_16(a_1_id_152, b_1_id_154);
  assert (((16u) 0) <=u a_1_id_156);
  return (a_0_id_155, a_1_id_156); }
```

Figure 4: Example of annotated source (top) and simplified Jasmin (bottom) programs.

Our goal of using CryptoLine to prove properties of Jasmin programs therefore raises three questions:

(1) How to construct a CryptoLine model of a Jasmin program?
(2) How to annotate the CryptoLine model of the Jasmin program?
(3) How to integrate the CryptoLine proof in higher-level proofs.

We now address each of these questions in turn.

Constructing the CryptoLine model. CryptoLine models must satisfy a set of strict requirements, such as being loop and branch free, to be correctly typed wrt to bit vector sizes and variable signedness, and to satisfy a notion of safety that, intuitively, means that there are no integer overflows in any of the bit vector computations.

The typical CryptoLine approach to creating a model is semi-automated, but it is not formally verified. One first creates a .gas file using a debugging tool, which is essentially a trace of the execution of assembly instructions where all operands have been resolved to either registers or constants (including memory addresses). This sequence is straight-line, and it provides enough information that permits rewriting the .gas file (almost 1-line-to-1-line) using an automatic script to obtain the model. This rewriting script manually is created for each proof, and is part of the proof process.

[2]https://github.com/openssl/openssl/blob/master/crypto/ec/asm/x25519-x86_64.pl
[3]https://github.com/fmlab-iis/cryptoline/blob/master/examples/openssl/x25519/x86_64/x25519_fe64_sqr.cl
[4]For a more detailed explanation of the CryptoLine semantics, please refer to Section A

The above CryptoLine methodology has the advantage of being source-language-agnostic. However, for our purposes, the source language is fixed to Jasmin, and Jasmin is supported by a formally verified compiler. We therefore propose an alternative CryptoLine model-generation process where we extend the Jasmin compiler, and its proof, to establish a formal connection between Jasmin programs and their corresponding CryptoLine models and proofs. To this end, we created in the Jasmin compiler a new compilation path that can convert a restricted class of Jasmin programs—which may include high-level control-flow, such as unrollable for loops, integer variables, arrays of registers and stack positions, etc.—into a lower level format that can be pretty-printed as a CryptoLine program. In what follows, we refer to the Jasmin program we want to prove correct as *source* and to the transformed program that is pretty-printed to CryptoLine as *simplified.* Figure 4 shows a simple example of the source and simplified Jasmin programs. The source program uses arrays and for loops. The resulting target simplified program is very close to a CryptoLine program, in that all high-level features were removed: the program is straight-line, it uses no arrays, it is static single-assignment. One goal of our approach is to minimize human intervention in constructing the CryptoLine model and proof. We therefore fix two default models of Jasmin programs (for computations over signed and unsigned integers) and require user intervention only when it is necessary to locally change the modeling of specific instructions via translation directives inserted at the source level. The details are given in Section 6. Furthermore, as we discuss next, the annotations required for the proof can be added directly to the source Jasmin program.

Annotating Jasmin programs. We extended the Jasmin compiler with support for an extensible annotation language that supports preconditions, post-conditions, as well as assertions that refer to the program state during evaluation. Figure 4 shows also examples of these annotations in the source program, and how the compiler translates them into the simplified program. Note, in particular the use of high-level big operators such as all to have a compact syntax for the predicates over the program state, and how these are removed in the annotations of the simplified program. In the following sections we will give the formal details about why this means that we can perform a proof over a simplified program, and still have the proof apply to the Jasmin source we are interested in. Here we just give the general intuition. In short, the compiler proof has been enriched to cover an extended version of the semantics that is *annotation aware*: the value of all assertions are registered by the semantics in a trace during program evaluation. Moreover, this proof of correctness provides a stronger semantic *equivalence* relation between source and simplified programs that allows reasoning about correctness in either of the programs interchangeably.

EasyCrypt semantics of a CryptoLine proof. Using the above extensions to the Jasmin compiler, we can already construct CryptoLine proofs of Jasmin programs. However, our goal is to be able to use such results in larger proof developments. For example, in this paper, while we use CryptoLine to prove correctness of leaf functions that carry out X25519 multi-precision arithmetic, we use these results in a larger proof of correctness for the full X25519 implementation, which in turn is integrated into the proof of correctness of the X-Wing hybrid KEM construction in Jasmin.

Our final contribution is a new extraction of annotated Jasmin programs to EasyCrypt that exports the semantics of a CryptoLine proof in a format that can be composed with other proofs. Specifically, we formalize the meaning of a CryptoLine proof in terms of the Jasmin semantics, and create in EasyCrypt an exact description of what CryptoLine has proved in terms of these semantics. This description leaves as proof goals to be discharged interactively in EasyCrypt any unproved assumptions made in the CryptoLine proofs, thereby enabling a flexible splitting of proof effort between the two tools. For the X-Wing example, the proof goals that arise in EasyCrypt to complete the CryptoLine proofs are the justification of a common technique in CryptoLine: asserting under bit-string semantics that a bit is 0, and then assuming an algebraic predicate that is implied by this fact over the integers. Discharging these goals in EasyCrypt is simple, and can be seen as machine-checking what the human user was already able to check by direct inspection. The examples we present at the end of the paper illustrate additional ways in which we can modularize a proof to better take advantage of the strengths of interactive and automatic proof techniques.

3 Annotated Jasmin Programs

We have extended the Jasmin syntax and semantics to allow for two types of annotations. Function declarations can be annotated with pre- and post-conditions using the keywords requires and ensures, and *annot* statements can be added anywhere in the code. All annot statements, $annot(k, e)$, have an associated kind assert or assume and a boolean expression. For simplicity, we write assert e for $annot(\text{assert}, e)$ (resp. for assume). We have also extended the syntax of Jasmin expressions with a generic (high order) fold operator over integers, which allows to easily encode big operators like all and sum, and we also allow the user to extend the language with abstract type and abstract operators. These operators can only be used in annotations, since they have no semantics and the compiler does not know how to compile them to assembly.

Semantics of an annotated program. To account for annotations, the semantics of Jasmin programs have been enriched in the Coq development. The evaluation of a Jasmin program defines, not only a relation between an initial state and a final state, but also the construction of a *trace t* keeping track of the annotations evaluated during the program execution. The trace is simply a list of pairs kind/predicate value. Formally, an *annot* statement leaves the state of the program unchanged, but it updates the trace with the kind and Boolean value of the associated expression in the current state.

$$\frac{[\![e]\!](s) = \mathrm{b}}{annot(k, e), (s, t) \Downarrow (s, t \mathbin{+\!\!+} [(k, b)])}$$

The semantics of a function call are also modified as shown in Figure 5. They add to the trace the kinds and values of the pre- and post-condition in two steps: the caller asserts the pre-condition before entering the callee, and assumes the post-condition upon return. The callee assumes the precondition upon entry, and asserts the post-condition before returning.

Finally, in what follows we will restrict our attention to *safe* Jasmin programs, which we define as follows.

$$\frac{[\![p(f)_{pre}]\!](\mathrm{v_a}) = \mathrm{b} \qquad p(f)_c, (\emptyset[p(f)_{param} := \mathrm{v_a}], [(\mathsf{assume}, b)]) \Downarrow^* (s', t) \qquad [\![p(f)_{res}]\!](s') = \mathrm{v_r} \qquad [\![p(f)_{post}]\!](\mathrm{v_a}, \mathrm{v_r}) = \mathrm{b'}}{f, \mathrm{v_a} \Downarrow^{p}_{call} \mathrm{v_r}, t + [(\mathsf{assert}, b')]}$$

$$\frac{a, s \Downarrow \mathrm{v_a} \qquad [\![p(f)_{pre}]\!](\mathrm{v_a}) = \mathrm{b} \qquad f, \mathrm{v_a}, \Downarrow^{p}_{call} \mathrm{v_r}, t \qquad [\![p(f)_{post}]\!](\mathrm{v_a}, \mathrm{v_r}) = \mathrm{b'} \qquad t'' = t + [(\mathsf{assert}, b)] + t' + [(\mathsf{assume}, b')]}{(r = f(a), (s, t)) \Downarrow ((s[r := \mathrm{v_r}]), t''}$$

Figure 5: Semantics of call in Jasmin with annotations

Definition 1 (Safe program). *We say a Jasmin program c is safe when $\forall s. \exists s', t. (c, (s, \epsilon)) \Downarrow^* (s', t)$.*

4 Certified compilation with traces

Our interaction with CryptoLine requires a simplified Jasmin program, where all arrays, if statements and for loops have been removed in order to obtain a straight-line, static single-assignment program that can be pretty-printed to the CryptoLine language in a trivial way. To do so we have created inside the Jasmin compiler a new compilation pipeline that is specific to CryptoLine and that removes all high-level features from an input Jasmin program. We will call this the *CryptoLine version* of the Jasmin compiler and, when clear from the context, the *program simplification* performed by the Jasmin compiler. This pass rejects Jasmin programs that include features not supported by our CryptoLine backend, such as direct memory access and array accesses that cannot be resolved at compile time. The simplification unrolls loops, replaces arrays with variables, and propagates constants (including those stored in global tables). Furthermore big operators in annotations are removed by partial evaluation. The final program is a valid annotated Jasmin program in static single-assignment form. The proof of the compiler (CryptoLine version) has been extended to take into account traces.

Theorem 1 (Preserving Trace). *The simplification performed by the Jasmin compiler preserves traces:*

$$\forall \sigma, s, t.\ (c, \sigma) \Downarrow^* (s, t) \implies \exists s'.\ (\mathcal{T}(c), \sigma) \Downarrow^* (s', t)$$

Where $\mathcal{T}(c)$ denotes the compilation of a program c.

A well known fact is that, because the semantics of Jasmin is deterministic, the above theorem implies the contraverse for safe programs.

Theorem 2. *The simplification performed by the Jasmin compiler reflects traces of safe programs.*

$$\forall \sigma, s, t.\ (\mathcal{T}(c), \sigma) \Downarrow^* (s, t) \implies \exists s'.\ (c, \sigma) \Downarrow^* (s', t)$$

Trace validity. In our theoretical framework we will also need the notions of trace and sub-trace validity.

Definition 2 (Valid Trace). *A trace t is valid, written $valid(t)$, when all the Boolean values in the trace are true (written $\top$).*

A CryptoLine proof does not guarantee full trace validity, but instead a restricted notion of sub-trace validity that we formalize in Section 6. Intuitively, CryptoLine guarantees that the assert subtrace is valid in the restricted set of executions where the assume trace holds.

Definition 3 (Valid Sub-trace). *We say the sub-trace of trace t of kind k is valid, written $valid(t)_\mathsf{k}$, when $valid(t)_\mathsf{k}$ is a derivation of the following rules:*

$$\frac{}{valid(\epsilon)_\mathsf{k}} \qquad \frac{valid(t)_\mathsf{k}}{valid((\_, \top) :: t)_\mathsf{k}} \qquad \frac{k' \neq \mathsf{k}}{valid((k', \bot) :: \_)_\mathsf{k}}$$

The intuition of this notion of validity is that, when checking for kind k, we are only interested in validity for executions in which annotations of the other kinds evaluate to true. When this is *not* the case, then validity trivially holds. In fact, this derivation is essentially encoding an implication: $valid(t)_\mathsf{k}$ holds if either some annotation of kind $k' \neq \mathsf{k}$ is false, or all annotations of kind k are true. An important property of sub-trace validity is completeness, which we prove in EasyCrypt for the concrete case of assume/assert.

Theorem 3 (Complete validity splitting).

$$valid(t)_\mathsf{assert} \wedge valid(t)_\mathsf{assume} \implies valid(t)$$

5 Methodology

So far we have discussed the meaning of an annotated program, so now we are ready to discuss our methodology to prove properties of Jasmin programs by relying on annotations.

To this end, we introduce two notions of correctness for Jasmin programs; one that permits establishing a weaker notion of partial correctness, and another one that permits reasoning about our target notion of total correctness.

Definition 4 (Well annotated program for a kind). *For given program c, and arbitrary predicates P and Q, we say a program is well annotated for kind* k *when:*

$$\{P\}c\{Q\}_\mathsf{k} \triangleq$$
$$\forall s, s', t. P(s) \implies (c; \mathsf{assert}\ Q, (s, \epsilon)) \Downarrow^* (s', t) \implies valid(t)_\mathsf{k}$$

We say a program is well annotated when the implication holds for $valid(t)$, i.e., execution always generates a full valid trace.

Intuitively, for the particular case of programs annotated only with assume and assert stements, a partially well annotated program is guaranteed to satisfy all assertions (including the post-condition) but only in executions where the assume annotations turn out to be true. Indeed, such a result says nothing about the executions in which the pre-condition and internal assume statements turn out to evaluate to false.

The stronger notion of well-annotated programs requires all annotations to be valid whenever the program starts from a state in which the precondition holds. Clearly, for safe programs, a well-annotated program is also totally correct in the standard sense:

Definition 5 (Total correctness).

$$[P]c[Q] \triangleq \forall s. P(s) \implies \exists s', t. (c, (s, \epsilon)) \Downarrow^* (s', t) \wedge Q(s')$$

Theorem 4. *A well annotated safe program is totally correct.*

The proof of this theorem is trivial, since reaching a final state is guaranteed by safety, and a well annotated program is known to satisfy the post-condition due to the assertion over the final state.

Methodology. Our methodology relies on the fact that we decompose a proof of functional correctness by 1) proving that a simplified version of the program is partially well annotated for asserts using CryptoLine (this first part is done fully automatically); and 2) proving that the source program is partially well annotated for assumes using EasyCrypt. This latter part is not always necessary, but it allows us to flexibly split the proof effort between CryptoLine and EasyCrypt, which is useful in several cases as we will illustrate with our examples. Formally, the soundness of our approach is captured by the following theorem.

Theorem 5 (Soundness). *Let c be an annotated Jasmin program and let $\mathcal{T}(c)$ denote its simplified counterpart. Then following conditions imply that $[P]c[Q]$ holds:*

(1) c is safe
(2) $\{\mathcal{T}(P))\}\ \mathcal{T}(c)\ \{\mathcal{T}(Q))\}_{\mathsf{assert}}$
(3) $\{P\}c\{Q\}_{\mathsf{assume}}$

Proof. By Theorem 2 we know that (2) implies $\{P\}\ c\ \{Q\}_{\mathsf{assert}}$. This, combined with (3) allows us to apply Theorem 3 and derive that the program is well annotated. Finally, we complete the proof by applying Theorem 4. □

6 Proving Partial Correctness using CryptoLine

We now explain how we use CryptoLine as a decision procedure for the partial correctness of simplified Jasmin programs.

CryptoLine and its semantics semantics. The semantics of CryptoLine and the meaning of a CryptoLine proof are described formally in Appendix A. In short, CryptoLine programs are straight-line and static single-assignment, just like simplified Jasmin programs. The main difference lies in the fact that CryptoLine variables are declared as words of fixed size, and they have a type annotation that indicates how they should be interpreted as an integer: they are either signed or unsigned. The semantics of CryptoLine instructions is defined by interpreting the values as integers, performing a computation over the integers, and injecting the result back into the type. The CryptoLine notion of safety requires proving that all such injection operations performed by the semantics are overflow-free (this depends on signedness). Otherwise the semantics raises an error. Like simplified Jasmin programs, CryptoLine programs are annotated with pre-conditions and post-conditions, and they can have intermediate assert and assume statements. The semantics of CryptoLine evaluates to an error whenever an assert statement evaluates to false. For assume statements there is no error: the execution of the program is blocked. The following notion of correctness, captures the meaning of successful CryptoLine proof [10, 14].

Definition 6 (CryptoLine correctness). *We say a program c is correct wrt CryptoLine for predicates P and Q if:*

$$\{P\}c\{Q\}_{\mathsf{CL}} \triangleq \forall s, s'.\ P(s) \implies (c, s) \Downarrow^*_{\mathsf{CL}} s' \implies s' \neq \mathsf{error} \land Q(s')$$

From simplified Jasmin to CryptoLine. We construct a pretty-printing function $\mathcal{P}$ from an annotated (simplified and safe) Jasmin program $c := [c_1, \ldots, c_n, \mathsf{return}\ v]$ that, by construction, has the following properties.

- State equivalence: there is a one-to-one correspondence between the variables in the Jasmin program and the subset of variables in the pretty-printed program that can occur in pretty-printed CryptoLine predicates. In what follows, when we use the same state to evaluate c and $\mathcal{P}(c)$ we mean that these variables have the same value as bitstrings in both programs.
- Predicate equivalence: if P is a predicate and s is a state, then $P(s) \equiv (\mathcal{P}(P))(s)$.
- Semantic equivalence for instructions: if c_i denotes a Jasmin statement that executes a Jasmin processor instruction, then $(\mathcal{P}(c_i), s) \Downarrow_{\mathsf{CL}} s' \implies s' \neq \mathsf{error} \implies (c_i, s) \Downarrow s'$
- We note that both simplified Jasmin programs and CryptoLine programs are static single-assignment, which means that each statement updates the state by creating new variables and assigning values to them. This allows for straightforward solutions when handling function calls. If c_i a Jasmin statement that calls function g, then $\mathcal{P}(c_i) := \mathsf{assert}(\mathcal{P}(pre(g))); \mathsf{assume}(\mathcal{P}(post(g)))$.

This last rule is very important because it allows to not always inline functions during the translation to CryptoLine, and allows for a modular proof where each function is proved only once. The following result allows us to use CryptoLine as a decision oracle for point (2) in Theorem 5.

Theorem 6. *Let c be a simplified safe Jasmin program and P Q be arbitrary predicates. Then, if $\mathcal{P}$ guarantees the properties stated above, we have:*[5]

$$\{\mathcal{P}(P)\}\mathcal{P}(c)\{\mathcal{P}(Q)\}_{\mathsf{CL}} \implies \{P\}c\{Q\}_{\mathsf{assert}}$$

Proof. We note that Jasmin programs do not contain recursive functions. Thus we can do the proof by induction using a lexicographical order on the call graph of c and the size of c. If c is empty it is trivial. If c is $i; c'$ where i is a processor instruction or an assert the result follows trivially by induction. If c is an $\mathsf{assume}(e); c'$, and if e evaluates to $\top$ the result follows by induction; if e evaluated to $\perp$ then the trace exists (because c is safe) and is valid (assert) because it includes $(\mathsf{assume}, \perp)$. If c is a function call followed by c', $\mathcal{P}(c)$ does not really evaluate the function: instead it simply introduces new variables and assumes the post condition. We now must refer to Figure 5 and observe that the callee Jasmin function assumes the pre-condition (which we already establish to be true), so validity is preserved. Also, since we know that the subroutine is well-annotated for asserts and we are in a state where the pre-condition holds, the trace produced by the subroutine can be appended to the previous trace with the guarantee that validity holds overall. Finally, we use the fact that the subroutine asserts the post-condition on the Jasmin side to derive that assuming the state resulting from the call return is covered by the proof of CryptoLine correctness of c. This means that we can take the Jasmin state resulting from the evaluation of the call and use it on the CryptoLine side in order to proceed the proof by evaluating the two programs in lockstep. □

[5]Note that in the context of Theorem 5 we will be instantiating this theorem with $(c, P, Q) := (\mathcal{T}(c'), \mathcal{T}(P'), \mathcal{T}(Q'))$ for some Jasmin source (c', P', Q').

```
fn _ec2(reg u64 g j) → (reg u64, reg u64)
requires #[prover=smt] { g == j}
ensures #[prover=smt] { result.0 == g + j}
{
#[kind=Assert, prover=smt] assert ( g == j);
 g = g + j;      return (g,g); }

fn _ec2 (reg u64 g_id_130, reg u64 j_id_131) → (reg u64, reg u64)
requires #[prover=smt] {(g_id_130 ==64u j_id_131)}
ensures #[prover=smt] {(g_id_135 ==64u (g_id_130 +64u j_id_131))}
{
 reg u64 g_id_135;
 #[kind="Assert", prover="smt"]
 assert (g_id_130 ==64u j_id_131);
 (_ , _ , _ , _ , _ ,g_id_135) = #ADD_64(g_id_130, j_id_131);
 return (g_id_135, g_id_135); }
```

Figure 6: Annotated Jasmin (top) and simplification (bottom).

Pretty-Printing Instructions. We will explain the pretty-printing using simple examples. Consider the following simplified Jasmin program, that corresponds to the source in Figure 8.

The pretty printing of the addition instruction generates the following string:

```
adds NONE_____146@uint1 g_135@uint64 g_130@uint64 j_131@uint64;
```

Note the introduction of the temporary variable to handle the carry. This is the default translation; alternatively, it is possible to annotate the source code with a modifier that pretty-prints the translation as:

```
add g_135@uint64 g_130@uint64 j_131@uint64;
```

However, in this case CryptoLine will reject the program as unsafe in case the carry cannot be proved to be zero. Nevertheless, note that in both cases the final value of the output variable is identical to the value it will take in the Jasmin semantics.

We follow the same strategy to model all Jasmin instructions of interest. Furthermore, we do so for two pretty-printing modes: unsigned (shown in the example above) and signed. The difference between the two is whether translated Jasmin variables (for which the notion of signedness is not defined) are translated to signed or unsigned CryptoLine variables.[6] For many operations, the pretty-printing is actually the same independently of signedness. However, for some instructions, such as arithmetic shifts, one needs to take care to ensure that the final value computed by the CryptoLine translation (assuming the program is declared safe) matches the one computed by Jasmin. As an example, the translation of assigning a constant 65535 to a 16-bit variable a in the signed case is `mov a_144@sint16 (-1)@sint16`.

Pretty-Printing Predicates. The pretty-printing of Jasmin predicates to CryptoLine has several interesting aspects. The first aspect is how we handle the distinction between CryptoLine algebraic and range goals. An attribute can be associated with each annotation to tag it as being intended for pretty-printing as an algebraic or range predicate, which allows $\mathcal{P}$ to perform a few checks. These include confirming that only an appropriate subset of Jasmin expressions can occur in each one of them (expressions computed over the integers for algebraic predicates vs expressions computed over Jasmin words/bit vectors for range predicates). For algebraic predicates we enrich the language of expressions using a few abstract operators that permit capturing the CryptoLine semantics of the annotations. We describe some examples:

- u16i, u32i, etc. are abstract predicates mapping Jasmin words to integers. They represent the integer interpretation of a Jasmin word.
- eqmod_int and eqmod are abstract predicates that represent congruences over the integers and polynomials, respectively.
- mon, mon0 are constructors for monomials, which one can use to construct polynomials.

We note that the Jasmin semantics says nothing about the meaning of these abstract operators, but the pretty-printing to CryptoLine actually assigns to them a semantics. For example u16i has the semantics of interpreting a Jasmin variable as a signed integer when using the signed mode of pretty-printing, and as an unsigned integer otherwise. Similarly, eqmod_int(a, b, c) is printed to the eqmod(a, b, c) CryptoLine operator, which over the integers has the semantics of $a - b = 0 \pmod{c}$.

Implications. We now shortly explain how the results in this section relate to the theoretical framework introduced in the previous sections. Consider some Jasmin function that is pretty-printed to CryptoLine and accepted as correct (and similarly for all functions in the call tree). Then we know that the program is well annotated for assertions, but compared to the previous section we have the novelty that these assertions now have the full semantics assigned by $\mathcal{P}$ and CryptoLine to the abstract operators that occur in the Jasmin code. We can therefore apply Theorem 5 and conclude that the semantics of the above result is reflected back to the Jasmin source and, furthermore, that to transform the CryptoLine proof into a total correctness result, it remains only to prove that the program is well annotated for assumes. (If no assume statements exist, then there is nothing left to prove.)

In the following section we explain how we recover the full semantics of the CryptoLine proof in EasyCrypt and, if necessary, discharge the side-conditions concerning assume statements. For this we rely on a simple EasyCrypt library that brings to EasyCrypt the concrete semantics of the abstract operators fixed by $\mathcal{P}$ and CryptoLine. The fact that these semantics are correct is part of our TCB. However, this assumption is easy to validate by human inspection, since the library is very small. For example, Figure 7 shows the relevant parts for the operators above.

7 Trace Semantics in EasyCrypt

We have implemented a dedicated extraction feature in the Jasmin compiler tool chain, which takes as input an annotated Jasmin program and generates an EasyCrypt file that captures the semantics of what was proved and left as an unproved assumption by CryptoLine. The extraction should be used when all functions in the call tree of the entry-point annotated Jasmin function have been approved by CryptoLine. Specifically, all the call tree is extracted

[6]We could have generalized this part of the translation and considered keeping signedness information for each Jasmin variable, but this is not needed for the examples we are interested in here.

```
op u16i = W16.to_sint. (* for proofs using signed interpretation *)
op u16i = W16.to_uint. (* for proofs using unsigned interpretation *)

op eqmod_int(a b m : int) : bool = (a−b) %% m = 0.

clone export Poly.Poly as IntPoly with
  type coeff = int.

abbrev (^) = PolyComRing.exp.

op mon(ind deg c : int) = c ** X ^ deg.
op mon0(c : int) = c ** X ^ 0.

op eqmod(a b : poly, c : poly list) =
   ∃  (ws : poly list), size ws = size c ∧ foldr (fun (xmi : poly*poly) (acc : poly) ⇒
        xmi.`1 * xmi.`2 + acc) poly0 (zip ws c) = (a − b).
```

Figure 7: EasyCrypt semantics of CryptoLine predicates.

```
proc _ec2 (g:W64.t, j:W64.t) : ((W64.t * trace) = {
  var old_j, old_g 4.t;        var trace__ec2:trace;
  old_g ← g;  old_j ← j;  trace__ec2 ← [];

  (* The precondition is assumed *)
  trace__ec2 ← (trace__ec2 ++ [(Assume, (g = j))]);

  (* The assertion is asserted *)
  trace__ec2 ← (trace__ec2 ++ [(Assert, (g = j))]);
  g ← (g + j);

  (* The postcondition is asserted *)
  trace__ec2 ← (trace__ec2 ++ [(Assert, (g = (old_g + old_j)))]);
  return (g), trace__ec2);
}
```

Figure 8: EasyCrypt extraction with traces.

to EasyCrypt, and it explicitly computes the trace of annotations as defined in the Jasmin semantics. Figure 8 shows the result of extracting the Jasmin function from Figure 6. The case of function calls is handled also as expected: all functions add their pre-condition as an assumption and assert the pre-conditions of subroutines. Similarly, each function asserts its own post-condition before returning, and assumes the post-condition of subroutines after they return. Note that the EasyCrypt extraction maintains the calls to subroutines, so the traces computed in EasyCrypt exactly match those described in Figure 5. The statement verified by CryptoLine translated to EasyCrypt as follows.

```
lemma _ec2_assert _g _j :
  hoare [M._ec2 : (((_j = j) ∧ (_g = g)) ∧ (_g = _j)) ⇒  (validk Assert (trace res))].
```

These lemmas are admitted with a comment indicating that they should be proved using CryptoLine. The statement that remains to be verified, corresponding to the assumed conditions in CryptoLine is stated as follows:[7]

```
lemma _ec2_assume_ _g _j :
    hoare [M._ec2 : ((_j = j) ∧ (_g = g)) ⇒  ((_g = _j) ⇒  (validk Assume (trace res)))].
```

This proof needs to be completed in EasyCrypt, and it is typically boilerplate, unless the assumes statements in the proof indeed require deep interactive reasoning in EasyCrypt (e.g. as we will see in the next section this can happen because a function was simply assumed to be correct in the CryptoLine proof introducing an assumption of the post-condition prior to returning). We then generate two lemmas, with synthethised EasyCrypt proofs, that permit deriving the total correctness result stated in Theorem 5. To be able to show that the function contract is holding, we need an additional lemma stating that the postcondition is in the trace :

```
lemma _ec2_valid_post _g _j : hoare [M._ec2 : ((_j = j) ∧ (_g = g)) ⇒
    ((valid (trace res)) ⇒  (res.`1.`1 = (_g + _j)))].

lemma _ec2_spec _g _j : hoare [M._ec2 :
    (((_j = j) ∧ (_g = g)) ∧ (_g = _j)) ⇒  (res.`1.`1 = (_g + _j))].
```

The first lemma establishes that a valid trace implies the post-condition, and the second lemma combines all previous proofs to get the total correctness hoare triple.

8 Examples

In this section we illustrate the formal verification methodology for Jasmin programs that combines the traditional EasyCrypt-based approach with the Jazzline approach we introduce in this paper.

8.1 X25519 and X-Wing

Figure 9 shows a snippet of the EasyCrypt specification of X-Wing: the procedure that expands the X-Wing secret key. This specification, makes calls to the EasyCrypt specifications of three lower-level cryptographic primitives: SHAKE-256, X25519 and ML-KEM. The figure also shows the specification of the X25519 curve operations.

We prove equivalence of the X-Wing implementation in Jasmin to this specification. The proof of equivalence relies directly on the relational Hoare logic provided by EasyCrypt and it is done in multiple hops, i.e., we prove equivalence across a sequence of implementations and use transitivity to derive the result we need: this expresses an equivalence between an extraction of the full X-Wing code to EasyCrypt (a single EasyCrypt module) and the X-Wing EasyCrypt specification. The first transformation proves equivalence of the extracted implementation to another one in which all the code corresponding to SHA-3, X25519 and ML-KEM is removed and is replaced to calls to the EasyCrypt modules over which correctness of SHA-3, X25519 and ML-KEM proofs has already been carried out. This proof is straightforward, as EasyCrypt can detect that the code used by X-Wing is syntactically identical to the ones over which the proofs were carried out. We carry out another hop to lower the level of abstraction of the X25519 specification and replace the functional operators that carry out the X-Wing operations with calls to procedures that implement the same operations imperatively—our specification of X25519 has both variants for convenience, and they are proved to be equivalent to enable using them interchangeably. Once these steps are performed, the top-level code of the X-Wing implementation is making calls to procedures in the exact same points as the X-Wing specification. This means that the proof of correctness can be performed by simply using EasyCrypt's relational hoare logic to argue that equivalence follows from the fact that the called procedures on both sides are themselves equivalent. For the ML-KEM and SHA-3 implementations these equivalences follow from the proofs carried out in [1, 3] and [2], respectively: our proof just applies the EasyCrypt correctness lemmas that were established in these pre-existing developments.[8] The approach introduced in this paper was instrumental in obtaining proofs for the X25519 implementations with minimal effort, as we describe next.

[7]Observe the encoding of the precondition of the function as an implication in the post-condition of the Hoare triple. This clearly equivalent formulation allows us to reduce the boiler-plate EasyCrypt code associated with function calls in CryptoLine proofs.

[8]Some effort was required to update these pre-existing proofs to the more recent version of EasyCrypt and streamline the modular composition of proofs, but these changes were boilerplate refactorings.

```
op spec_decode_scalar_25519 (k:W256.t) =
  let k = k[0   ⟵ false] in
  let k = k[1   ⟵ false] in
  let k = k[2   ⟵ false] in
  let k = k[255 ⟵ false] in
  let k = k[254 ⟵ true ] in k.

op spec_decode_u_coordinate (u:W256.t) = let u = u[255 ⟵ false] in u.

op spec_add_and_double (qx : zp) (nqs : (zp * zp) * (zp * zp)) =
  let x_1 = qx in
  let (x_2, z_2) = nqs.`1 in
  let (x_3, z_3) = nqs.`2 in
  let a  = x_2 + z_2 in
  let aa = a * a in
  let b  = x_2 + (– z_2) in
  let bb = b*b in
  let e = aa + (– bb) in
  let c = x_3 + z_3 in
  let d = x_3 + (– z_3) in
  let da = d * a in
  let cb = c * b in
  let x_3 = (da + cb)*(da + cb) in
  let z_3 = x_1 * ((da + (– cb))*(da + (– cb))) in
  let x_2 = aa * bb in
  let z_2 = e * (aa + (inzp 121665 * e)) in ((x_2,z_2), (x_3,z_3)).

op spec_montgomery_ladder(init : zp, k : W256.t) =
  let nqs0 = ((Zp.one,Zp.zero),(init,Zp.one)) in
  foldl (fun (nqs : (zp * zp) * (zp * zp)) ctr ⟹
        if spec_ith_bit k ctr
        then spec_swap_tuple (spec_add_and_double init (spec_swap_tuple(nqs)))
        else spec_add_and_double init nqs) nqs0 (rev (iota_ 0 255)).
```

```
op spec_encode_point (q: zp * zp) : W256.t  =
   let q = q.`1 * (ZModpRing.exp q.`2 (p – 2)) in
       W256.of_int (asint q).

op spec_scalarmult_internal (k: zp) (u: W256.t) : W256.t =
    let r = spec_montgomery_ladder k u in
       spec_encode_point (r.`1).

op spec_scalarmult (k: W256.t) (u: W256.t) : W256.t =
   let k = spec_decode_scalar_25519 k in
   let u = spec_decode_u_coordinate u in
       spec_scalarmult_internal (inzp (to_uint u)) k.

op spec_scalarmult_base (k:W256.t) : W256.t =
   spec_scalarmult (k) (W256.of_int(9%Int)).

proc expandDecapsulationKey(sk : secretkey) : expandedSecretKey  = {
       var expanded, coins1, coins2, coins3, pk_M, sk_M, pk_X_256, pk_X, sk_X, k;

       expanded ⟵ SHAKE256_32_96 sk;

       coins1 ⟵ Array32.init (fun (i: int) ⟹  expanded[0 + i]);
       coins2 ⟵ Array32.init (fun (i: int) ⟹  expanded[32 + i]);
       coins3 ⟵ Array32.init (fun (i: int) ⟹  expanded[64 + i]);

       sk_X ⟵ coins3;
       pk_X_256 ⟵ scalarmult_base (W32u8.pack32 (to_list sk_X));
       pk_X ⟵ Array32.of_list W8.zero (W32u8.to_list pk_X_256);

      (pk_M, sk_M) ⟵ MLKEM.kg_derand(coins1, coins2);
       k ⟵ (sk_M, sk_X, pk_M, pk_X);
       return k;
   }
```

Figure 9: Snippets of the X-Wing and X25519 EasyCrypt specifications.

Proving X25519 in EasyCrypt. The correctness proof for curve X25519 was done simultaneously for two implementations: a reference implementation, and an optimized implementation using the mulx, adox and adcx family of x86-64 operation, that enable the interleaving of add-with-carry operations and multiplications. This allows us to showcase the advantages of using EasyCrypt for the high-level parts of such proofs: we are able essentially reuse the proofs for all the architecture-agnostic parts of the implementation, i.e., the non-leaf functions that are orchestrating the finite-field operations in the correct sequence to compute the elliptic curve operations.

Technically, the proofs of both implementations proceed roughly as we described for the X-Wing proof. We construct an EasyCrypt imperative specification of the functions that compute the curve operations and prove this equivalent to the functional counterpart. This now moves our functional correctness target to a specification of X25519 that has a procedure call-tree that is very close to that adopted by the Jasmin implementations. Indeed, all functions except the leaves are essentially managing calls to lower-level functions, and this structure is the same for both the reference and optimized Jasmin implementations.

The leaf functions perform the same computation in both implementations and therefore satisfy the same contract. However, the way in which the computation is carried out, and therefore the proofs, diverge. At this point, we identify those proofs that are algebraically rich, and for which closely matching CryptoLine proofs already exists. The exceptions are simple functions such as those that fix some bits in the representations of values and/or perform type conversions to/from byte arrays. These latter functions we prove directly in EasyCrypt, and the proofs are straightforward. This highlights another advantage of using EasyCrypt: we have access in the same tool and simultaneously to a bit-level semantics and to a high-level integer-based semantics of word operations, so we can choose the most convenient one to carry out a proof.

The remaining leaf functions comprise addition, subtraction, multiplication, squaring and reduction modulo 2^{255}-19. For all operations, the inputs and outputs are represented as 4x64-bit registers, and the contracts are of the general form shown below for the multiplication operation:

```
hoare [M.__mul4_rsr :
   inzpRep4 fs = _f ∧ inzpRep4 g = _g ⟹  inzpRep4 res = _f * _g
].
```

Here, inzpRep4 is an EasyCrypt operator that takes an array of four 64-bit words, reconstructs the represented multi-precision integer and injects it into the finite field $\mathbb{F}_{2^{255}-19}$. The contract states that the result represents an integer that is congruent to the product of the (integers represented by the) input values, modulo 2^{255}-19.

Using Jazzline to conclude the proof. We conclude the proof by creating an annotated version of each Jasmin function that we want to prove. We show snippets of the multiplication code in Figure 10. We can see on the right-hand side the contract expressing that the function guarantees the desired congruence, expressed here over the integers. On the left-hand side we can see one of the subroutines where we use the strategy of proving using range predicates (i.e., SMT) that a carry flag is zero, and then assume this same property in the algebraic side of the proof.

Throughout, we are using the default translation of instructions to CryptoLine and unsigned interpretation for all variables. CryptoLine accepts the proof, which means that we can extract the result to EasyCrypt and obtain a statement of this result as described in Section 7. The top-level EasyCrypt theorem, which we are able to derive with essentially no effort is shown in Figure 10 (right).

We note, however that this result applies to an EasyCrypt module that is instrumented with all the operational code required to capture the semantics of asserts and assumes, whereas we need the result to hold over the EasyCrypt extraction of the actual implementation that we are proving correct. However, this final result is trivial to obtain in EasyCrypt by proving that the two versions of the EasyCrypt code (the non-instrumented and the instrumented one) are, in fact, computing the same result. Again, EasyCrypt is able to easily determine that the parts of the code that carry out the computation are syntactically identical.[9]

Lastly, for some of the low-level procedures, such as the modular reduction leaf function, we illustrate another possibility for composable reasoning with CryptoLine proofs. The proof requires a case analysis, considering for each case that input is in a particular range, such as 0 and $2^{255} - 19$. Hence, we proved the function's correctness multiple times using CryptoLine, with different assert/assume annotations, capturing conditions that follow from each specific case, and that enable an automatic proof. We can then compose these proofs in EasyCrypt and obtain a result that covers the entire range of possible inputs.

8.2 The ML-KEM NTT

Cryptographic constructions based on polynomial rings often rely on the number theoretic transform (NTT) to speed-up multiplications of ring elements. Kyber and ML-KEM are no exception. They are constructed over the polynomial ring $\mathcal{R}_q := \mathbb{Z}_q[X]/(X^{256}+1)$, where $q = 3329$ is a small prime. Both input and output of NTT can be written as a sequence of 256 coefficients in $\mathbb{Z}_q$. Indeed, the NTT is usually specified as an in-place algorithm over such sequences, which is the way in which implementations are expected to perform the computation. In particular, the FIPS-203 document [13] specifies the algorithm as given in Figure 14 (left). A quick look at the right-hand side of Figure 14, which includes the Jasmin reference implementation of the NTT routine, shows that the high-level control flow of both routines is very close, and the main difference between specification and implementations lies in the fact that the former is manipulating polynomial coefficients, which are field elements, whereas the implementation is manipulating 16-bit words that represent the corresponding field elements. In the EasyCrypt proofs of ML-KEM implementations carried out in [1, 3] this closeness is explored to construct a direct equivalence proof between two programs in EasyCrypt (ML-KEM specification and model of the Jasmin program) using relational Hoare logic. The non-trivial part of that proof concerns the lower level functions fqmul and poly_reduce, which perform multiplication modulo q in Montgomery representation and coefficient-wise Barrett reduction, respectively.

Algebraic proofs of the NTT in CryptoLine. The above equivalence proof does not capture the mathematical semantics of the computed NTT function: intuitively, it just shows that the two EasyCrypt algorithms (specification and model of implementation) compute the same result. CryptoLine proofs of the ML-KEM NTT are expressed differently: they establish that the output of the NTT operation can be seen as 128 degree 1 polynomials, as given by the formula below, also extracted from FIPS-203.

$$(f \pmod{X^2 - \zeta^{2\mathsf{BitRev}_7(0)+1}}, \ldots, f \pmod{X^2 - \zeta^{2\mathsf{BitRev}_7(127)+1}})$$

Here, f represents the input polynomial, $\zeta = 17$, and $\mathsf{BitRev}(i)$ is the integer represented by bit-reversing the unsigned 7-bit value that represents $i \in [0, 127]$. This result is, on one hand, semantically richer and, on the other hand, straightforward to obtain fully automatically using CryptoLine, under the assumption that the lower level computations corresponding to the fqmul and poly_reduce routines mentioned above are correct. However, establishing the correctness of these lower level routines is not straightforward to derive automatically, except by brute-forcing the computation using SAT. This could be done for ML-KEM, but this does not scale well to schemes that use larger primes, such as Dilithium/ML-DSA. We use the above scenario as motivation to showcase the advantages of our approach.

Bridging imperative and algebraic NTT specifications. We create in EasyCrypt a general theory that relates imperative specifications of the NTT as given in FIPS-203, and used in the formal specification of ML-KEM in [1, 3], to the CRT-based algebraic specification used in CryptoLine. This result, combined with the algebraic post-condition established automatically by CryptoLine for the Jasmin code, yields an alternative proof of functional correctness for the NTT. Crucially, the proof effort required to establish the relation between the imperative and algebraic specifications can be amortized for other implementations of the ML-KEM NTT and for other polynomial-ring based constructions such as ML-DSA.[10] The listing in Figure 11 shows the four EasyCrypt lemmas that permit obtaining the desired result. The ntt_equiv lemma shows that the annotated Jasmin program Poly_ntt.M._poly_ntt that was proved using CryptoLine is equivalent to the Jasmin implementation for which we want the final theorem to hold Poly_ntt_orig.M._poly_ntt. Here the equivalence is not a syntactic identity because we needed to replace while loops with unrollable for loops, but the proof is nevertheless straightforward. The _poly_ntt_spec_p lemma is the total correctness version of the CryptoLine result, which we obtain simply by proving termination; again, a straightforward proof. Note that, from CryptoLine we not only get the algebraic specification, but also a range contract that sets the admissible bounds of inputs and the enforced bounds at the outputs. Finally, the main lemma ntt_correct is obtained as a consequence of the above lemmas, and the fact that the ntt operator that captures the semantics of imperative specification is equivalent to the algebraic specification by the result in lemma ntt_correct_core.

Proving parts of the call-tree in EasyCrypt. We show in Figure 12 how, in EasyCrypt, we can complete a CryptoLine proof of the NTT function in which the correctness of field multiplication and Barrett reduction subroutines was assumed. Concretely, lemma trans_reduce expresses a relational contract between function M.__barrett_reduce and function Poly_ntt_orig.M.__barrett_reduce. The

[9]We will see in the next example, however, that for some proofs relying on CryptoLine, it is useful to annotate a Jasmin program that is actually not trivially equivalent to the one for which the proof is intended.

[10]For example, we reuse this result for a CryptoLine-based proof of the AVX2 implementation of the ML-KEM NTT, as we will describe later in the paper.

```
inline fn __mul4_c1( reg u64[4] h r g, reg u64 f z, reg bool   cf of)
  → reg u64[4],reg u64[4],reg bool,reg bool {
 reg u64 hi lo;
 ( hi, lo )   = #MULX ( f, g[0] );
 of, h[1]     = #ADOX ( h[1], lo, of );
 cf, h[2]     = #ADCX ( h[2], hi, cf );
 ( hi, lo )   = #MULX ( f, g[1] );
 of, h[2]     = #ADOX ( h[2], lo, of );
 cf, h[3]     = #ADCX ( h[3], hi, cf );
 ( hi, lo )   = #MULX ( f, g[2] );
 of, h[3]     = #ADOX ( h[3], lo, of );
 cf, r[0]     = #ADCX ( r[0], hi, cf );
 ( r[1], lo ) = #MULX ( f, g[3] );
 of, r[0]     = #ADOX ( r[0], lo, of);
 cf, r[1]     = #ADCX ( r[1], z, cf);
 of, r[1]     = #ADOX ( r[1], z, of);
 #[kind=Assert, prover=smt] assert (¬cf);
 #[kind=Assume, prover=cas] assert (b2i(cf) == 0);
 #[kind=Assert, prover=smt] assert (¬of);
 #[kind=Assume, prover=cas] assert (b2i(of) == 0);
 return h, r, cf, of;
}
```

```
fn __mul4_rsr(stack u64[4] fs, reg u64[4] g) → reg u64[4]
 ensures #[prover=cas] {
  eqmod (
    \sum (ii ∈ 0:4) (pow(2, 64∗ii)∗u64i(result.0[ii])),
    \sum (ii ∈ 0:4) (pow(2, 64∗ii)∗u64i(fs[ii])) ∗
       \sum (ii ∈ 0:4) (pow(2, 64∗ii)∗u64i(g[ii])),
    single((pow(2,255)) − 19)
   ) }
{
  g0 = #copy(g);  z = 0;
  f = fs[0];
  h, r, cf, of = __mul4_c0(      f, g0, z, cf, of);
  f = fs[1];
  h, r, cf, of = __mul4_c1(h, r, f, g0, z, cf, of);
  f = fs[2];
  h, r, cf, of = __mul4_c2(h, r, f, g0, z, cf, of);
  f = fs[3];
  h, r, cf, of = __mul4_c3(h, r, f, g0, z, cf, of);
  _38 = 38;
  h = __reduce4(h, r, _38, z, cf, of);
  return h;
}
```

```
lemma __mul4_rsr_spec  :
    ∀  _fs _g,
    hoare [M.__mul4_rsr :
    (((_g = g) ∧ (_fs = fs)) ∧ true) ⇛
    (eqmod
    (foldr (fun x ⇒  (fun (acc: int) ⇒  (x + acc))) 0
    (map (fun ii ⇒  ((pow 2 (64 ∗ ii)) ∗ (u64i res.` 1[ii]))) (iota_ 0 4)))
    ((foldr (fun x ⇒  (fun (acc: int) ⇒  (x + acc))) 0
     (map (fun ii ⇒  ((pow 2 (64 ∗ ii)) ∗ (u64i _fs[ii]))) (iota_ 0 4))) ∗
    (foldr (fun x ⇒  (fun (acc: int) ⇒  (x + acc))) 0
    (map (fun ii ⇒  ((pow 2 (64 ∗ ii)) ∗ (u64i _g[ii]))) (iota_ 0 4))))
    (single ((pow 2 255) − 19)))].
```

Figure 10: Snippet of the Jazzline proof for the X25519 mulx multiplication (left, middle). EasyCrypt theorem (right).

```
lemma ntt_equiv :
 equiv [ Poly_ntt_orig.M._poly_ntt ~  Poly_ntt.M._poly_ntt : ={arg} ⇛  res{1} = res{2}.` 1 ].

lemma _poly_ntt_spec_p  _rp:
    phoare [Poly_ntt.M._poly_ntt : _rp = arg ∧
    foldr (fun (x acc : bool) ⇒  x ∧ acc) true (map (fun (ii : int) ⇒
          (W16.of_int ((−2) ∗ 3329) \sle arg[ii]) ∧
               (arg[ii] \slt W16.of_int (2 ∗ 3329))) (iota_ 0 256)) ⇛
    foldr (fun (x acc : bool) ⇒  x ∧ acc) true
     (map (fun (ii : int) ⇒  (W16.zero \sle res.` 1[ii]) ∧
               (res.` 1[ii] \slt W16.of_int (2∗3329))) (iota_ 0 256)) ∧
    foldr (fun (x acc : bool) ⇒  x ∧ acc) true (map (fun (pp : int) ⇒
      eqmod (foldr (fun (x acc : poly) ⇒  x + acc) (mon0 0)
       (map (fun (ii : int) ⇒  mon 0 ii (u16i _rp[ii])) (iota_ 0 256)))
        (mon0 (u16i res.` 1[pp ∗ 2]) + mon 0 1 (u16i res.` 1[pp ∗ 2 + 1])) (pair (mon0 3329)
            (mon 0 2 1 − mon0 (u16i Poly_ntt.zetas_power[pp])))) (iota_ 0 128))] = 1%r

lemma ntt_correct_core (_r : Zq.coeff Array256.t) (_rp p: W16.t Array256.t) :
   _r = lift_array256 _rp ⇒
   foldr (fun (x acc : bool) ⇒  x ∧ acc) true
    (map (fun (pp : int) ⇒  eqmod (foldr (fun (x acc : poly) ⇒  x + acc) (mon0 0)
           (map (fun (ii : int) ⇒  mon 0 ii (u16i _rp[ii])) (iota_ 0 256)))
         (mon0 (u16i p[pp ∗ 2]) + mon 0 1 (u16i p[pp ∗ 2 + 1])) (pair (mon0 3329)
         (mon 0 2 1 − mon0 (u16i zetas_power[pp])))) (iota_ 0 128)) ⇒  ntt _r = lift_array256 p.

lemma ntt_correct _r :
  phoare[M._poly_ntt :
     _r = lift_array256 rp ∧  signed_bound_cxq rp 0 256 2  ⇛  ntt _r = lift_array256 res ∧
        ∀  k, 0≤k<256 ⇒  bpos16 res[k] (2∗q)] = 1%r.
exlim rp ⇒  _rp; conseq ntt_equiv (_poly_ntt_spec_p _rp).
```

Figure 11: EasyCrypt lemmas for NTT functional correctnes.

latter function has been proved in EasyCrypt to satisfy the required contract—the name of this lemma is Fq.barrett_reduce_corr_h. The former function includes an annotated assume statement of the post-condition immediately before the return statement, which guarantees a trivial proof in CryptoLine.

The relational contract establishes a bridge between the known result for the returned value of Poly_ntt_orig.M.__barrett_reduce and the unproved assumption in CryptoLine. Here, we leverage the fact that the semantics of CryptoLine assert and assume statements are captured by global variables in EasyCrypt: this allows us to refer to predicates that are evaluated in the middle of the annotated program (in this case just before the return statement) and relate them to the outputs of the program we have proved in EasyCrypt.

Concretely, the trans_reduce lemma says that, if both functions start from the same input, then both will produce the same result but, moreover, the flag that encodes the assumed predicate res1.`2.`3 is equivalent to the post-condition over the result. This permits deriving a proof for the assumed predicate in lemma __barrett_reduce_assume

```
equiv trans_reduce _a : M.__barrett_reduce ~  Poly_ntt_orig.M.__barrett_reduce :
 ={arg} ∧ to_sint arg{1} = _a
  ⇛  res{1}.` 1 = res{2} ∧
    (validk Assume res{1}.` 2 ⇔  (((((W16.of_int 0) \sle res{1}.` 1) ∧
      (res{1}.` 1 \slt (W16.of_int (2∗3329))))) ∧ (eqmod_int (u16i res{1}.` 1) (_a) 3329))).

lemma __barrett_reduce_assume  :
    hoare [M.__barrett_reduce : true ⇛  (assume_proof_ res)].
proof.
exlim arg ⇒  _a.
conseq  (trans_reduce (to_sint _a)) (Fq.barrett_reduce_corr_h (to_sint _a)) .
```

Figure 12: EasyCrypt proof of leaf functions.

simply as a consequence of the above relation and the fact that the result of the Poly_ntt_orig.M.__barrett_reduce function has already been shown to be correct.

Extension to vectorized instructions. In Appendix B we describe further extensions to our generation of CryptoLine proofs from annotated Jasmin programs that allow us to take advantage of the vectorized extension to CryptoLine. The motivating example is the AVX2 implementation of the NTT, where we showcase the applicability of our general result for interchangeably using algebraic and imperative specifications of the transformation in other proofs. However, we also describe the needed extensions to annotations required refer to subwords in annotations, e.g., when a 256-bit word is seen by the vectorized implementation as sixteen 16-bit words, and how we extend the pretty-printing mechanism in a way that permits generating well-typed Cryptoline programs when the vectorized implementation may execute in sequence instructions that process sub-words of different sizes.

9 Conclusion

In this paper we show that it is possible to integrate two formal verification frameworks, in a sound way, to enable proofs that explore the advantages of both. This permits amortizing proof effort, and allows for collaborative work in proof development using different formal verification approaches. As illustration of this fact we present a functionally correct high-speed implementation of X-Wing in Jasmin, where we reused proofs initially done using CryptoLine. In future work we plan to extend the Jasmin compiler to cover a wider range of programs in simplification, and to explore the possibility of automatically transform vectorized instructions into scalar ones in a certified way.

References

[1] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Vincent Laporte, Jean-Christophe Léchenet, Tiago Oliveira, Hugo Pacheco, Miguel Quaresma, Peter Schwabe, Antoine Séré, and Pierre-Yves Strub. 2023. Formally verifying Kyber Episode IV: Implementation correctness. *IACR TCHES* 2023, 3 (2023), 164–193. https://doi.org/10.46586/tches.v2023.i3.164-193

[2] José Bacelar Almeida, Cecile Baritel-Ruet, Manuel Barbosa, Gilles Barthe, François Dupressoir, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Alley Stoughton, and Pierre-Yves Strub. 2019. Machine-Checked Proofs for Cryptographic Standards: Indifferentiability of Sponge and Secure High-Assurance Implementations of SHA-3. In *ACM CCS 2019*, Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz (Eds.). ACM Press, 1607–1622. https://doi.org/10.1145/3319535.3363211

[3] José Bacelar Almeida, Santiago Arranz Olmos, Manuel Barbosa, Gilles Barthe, François Dupressoir, Benjamin Grégoire, Vincent Laporte, Jean-Christophe Léchenet, Cameron Low, Tiago Oliveira, Hugo Pacheco, Miguel Quaresma, Peter Schwabe, and Pierre-Yves Strub. 2024. Formally Verifying Kyber - Episode V: Machine-Checked IND-CCA Security and Correctness of ML-KEM in EasyCrypt. In *CRYPTO 2024, Part II (LNCS, Vol. 14921)*, Leonid Reyzin and Douglas Stebila (Eds.). Springer, Cham, 384–421. https://doi.org/10.1007/978-3-031-68379-4_12

[4] Andrew W. Appel. 2015. Verification of a Cryptographic Primitive: SHA-256. *ACM Trans. Program. Lang. Syst.* 37, 2 (2015), 7:1–7:31. https://doi.org/10.1145/2701415

[5] Manuel Barbosa, Gilles Barthe, Karthik Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. 2021. SoK: Computer-Aided Cryptography. In *2021 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 777–795. https://doi.org/10.1109/SP40001.2021.00008

[6] Manuel Barbosa, Deirdre Connolly, João Diogo Duarte, Aaron Kaiser, Peter Schwabe, Karoline Varner, and Bas Westerbaan. 2024. X-Wing. *CiC* 1, 1 (2024), 21. https://doi.org/10.62056/a3qj89n4e

[7] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. 2011. Computer-Aided Security Proofs for the Working Cryptographer. In *CRYPTO 2011 (LNCS, Vol. 6841)*, Phillip Rogaway (Ed.). Springer, Berlin, Heidelberg, 71–90. https://doi.org/10.1007/978-3-642-22792-9_5

[8] Deirdre Connolly, Peter Schwabe, and Bas Westerbaan. 2024. *X-Wing: general-purpose hybrid post-quantum KEM.* Internet-Draft draft-connolly-cfrg-xwing-kem-06. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-connolly-cfrg-xwing-kem/06/ Work in Progress.

[9] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. 2019. Simple High-Level Code for Cryptographic Arithmetic - With Proofs, Without Compromises. In *2019 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 1202–1219. https://doi.org/10.1109/SP.2019.00005

[10] Yu-Fu Fu, Jiaxiang Liu, Xiaomu Shi, Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. 2019. Signed Cryptographic Program Verification with Typed CryptoLine. In *ACM CCS 2019*, Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz (Eds.). ACM Press, 1591–1606. https://doi.org/10.1145/3319535.3354199

[11] Joel Kuepper, Andres Erbsen, Jason Gross, Owen Conoly, Chuyue Sun, Samuel Tian, David Wu, Adam Chlipala, Chitchanok Chuengsatiansup, Daniel Genkin, Markus Wagner, and Yuval Yarom. 2023. CryptOpt: Verified Compilation with Randomized Program Search for Cryptographic Primitives. *Proc. ACM Program. Lang.* 7, PLDI (2023), 1268–1292. https://doi.org/10.1145/3591272

[12] Adam Langley, Mike Hamburg, and Sean Turner. 2016. Elliptic Curves for Security. RFC 7748. https://doi.org/10.17487/RFC7748

[13] National Institute of Standards and Technology. 2024. FIPS 203 – Module-Lattice-Based Key-Encapsulation Mechanism Standard. https://doi.org/10.6028/NIST.FIPS.203

[14] Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. 2017. Certified Verification of Algebraic Properties on Low-Level Mathematical Constructs in Cryptographic Programs. In *ACM CCS 2017*, Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM Press, 1973–1987. https://doi.org/10.1145/3133956.3134076

[15] Yi Zhou, Sydney Gibson, Sarah Cai, Menucha Winchell, and Bryan Parno. 2023. Galápagos: Developing Verified Low Level Cryptography on Heterogeneous Hardwares. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, 2023*, Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda (Eds.). ACM, 2113–2127. https://doi.org/10.1145/3576915.3616603

[16] Yi Zhou, Sydney Gibson, Sarah Cai, Menucha Winchell, and Bryan Parno. 2023. Galápagos: Developing Verified Low Level Cryptography on Heterogeneous Hardwares. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security* (Copenhagen, Denmark) *(CCS '23)*. Association for Computing Machinery, New York, NY, USA, 2113–2127. https://doi.org/10.1145/3576915.3616603

[17] Jean Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. 2017. HACL*: A Verified Modern Cryptographic Library. In *ACM CCS 2017*, Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM Press, 1789–1806. https://doi.org/10.1145/3133956.3134043

A CryptoLine formally

A program in the CryptoLine language is a model of a cryptographic program that can be used to prove functional correctness properties. The language was created to allow the verification of programs written in different source languages and architectures. We give only a summary of the main features and refer the interested reader to [10, 14] for the full details.

Syntax and type system. CryptoLine types are of the form $Type ::= \mathsf{uint}\ Width \mid \mathsf{sint}\ Width$, with $Width$ a positive integer. These types represent unsigned and signed bitwords of size *Width*. For width w, they are inhabitted by integers i in the range $0 \leq i < 2^w$ for uint and $-2^{w-1} \leq i < 2^{w-1}$ for sint, consistently with the standard unsigned and two's complement interpretations of bitword integer representations.

The type of a variable is defined by its declaration. Constant integer literals must also be annotated with a type, which means that they always represent a bitword. A CryptoLine program consists of variable declarations followed by instructions. A typical instruction retrieves values from sources/atoms and stores them in destinations/variables. An atom is either a variable or a constant. The syntax is given in Figure 13. Instructions include operations that are pervasive across architectures, such as addition and multiplication (there is no overloading of instruction names for signed and unsigned values); other instructions, such as splitting and joining bitwords to obtain bitwords of different sizes, can be used to model additional processor operations. Finally, assert and assume instructions can also occur at any point in a program, both expressed as the conjunction of two types of predicates: range predicates (*RPred*) and algebraic predicates (*APred*). Note that, for the purpose of this paper, CryptoLine programs are straight-line.

The CryptoLine type system is, for the most part, straightforward. Many operations, namely arithmetic instructions, are closed over the type (i.e. word size and signedness). Unsigned operations have the expected typing, and only for signed operations there are some important nuances that aim to guarantee consistency with the interpretation over the integers of the behavior of some operations. For example, splitting a signed bitword will result in a signed bitword for the most significant bits and an unsigned bitword for the least significant bits. For example, suppose one splits –1, represented as $(1, 1)$ in $\mathsf{sint}\ 2$, into two bitwords of size 1. This will result in –1 for the most significant bit, represented as 1 in $\mathsf{sint}\ 1$, and 1 for the least significant bit, represented as 1 in $\mathsf{uint}\ 1$. This guarantees, not only that is the bit-level semantics of split correct, but also that one can naturally view the relation over the integers between the inputs and outputs of the split instruction as

$$\mathbb{Z}[(-1)@\mathsf{sint}\ 2] = \mathbb{Z}[(-1)@\mathsf{sint}\ 1] * 2^1 + \mathbb{Z}[1@\mathsf{uint}\ 1]\,.$$

Semantics. The semantics of CryptoLine are standard when it comes to statements/instructions. A program state is formalized by an environment ϵ, a mapping from variables to bitwords. The interpretation of the value of a variable in ϵ depends on its type. The semantics of instructions are defined by interpreting the value of the bitword as an integer value, performing a computation over

$$
\begin{array}{lllllllll}
 & & \multicolumn{7}{l}{\mathit{Num} ::= \cdots \mid -2 \mid -1 \mid 0 \mid 1 \mid 2 \mid \cdots \quad \mathit{Const} ::= \mathit{Num}@\mathit{Type} \quad \mathit{Var} ::= \cdots \mid x \mid y \mid z \mid \cdots \quad \mathit{Atom} ::= \mathit{Var} \mid \mathit{Const}} \\
\mathit{Exp} & ::= & \mathit{Atom} & \mid & \mathit{Exp} + \mathit{Exp} & \mid & \mathit{Exp} - \mathit{Exp} & \mid & \mathit{Exp} \times \mathit{Exp} \\
\mathit{Apred} & ::= & \multicolumn{7}{l}{\mathit{Exp} = \mathit{Exp} \mid \mathit{Exp} \equiv \mathit{Exp} \ \mathrm{mod}\ \mathit{Exp} \mid \mathit{Apred} \land \mathit{Apred} \quad \mathit{Rpred} ::= \mathit{Exp} = \mathit{Exp} \mid \mathit{Exp} < \mathit{Exp} \mid \mathit{Rpred} \land \mathit{Rpred}} \\
\mathit{Inst} & ::= & \mathrm{mov}\ \mathit{Var\,Atom} & \mid & \mathrm{cmov}\ \mathit{Var\,Var\,Atom\,Atom} & \mid & \mathrm{cast}\ \mathit{Var@Type\,Atom} & & \\
 & \mid & \mathrm{uadd}\ \mathit{Var\,Atom\,Atom} & \mid & \mathrm{uadds}\ \mathit{Var\,Var\,Atom\,Atom} & \mid & \mathrm{uadc}\ \mathit{Var\,Atom\,Atom\,Atom} & \mid & \mathrm{uadcs}\ \mathit{Var\,Var\,Var\,Atom\,Atom} \\
 & \mid & \mathrm{sadd}\ \mathit{Var\,Atom\,Atom} & \mid & \mathrm{sadds}\ \mathit{Var\,Var\,Atom\,Atom} & \mid & \mathrm{sadc}\ \mathit{Var\,Atom\,Atom\,Atom} & \mid & \mathrm{sadcs}\ \mathit{Var\,Var\,Var\,Atom\,Atom} \\
 & \mid & \mathrm{usub}\ \mathit{Var\,Atom\,Atom} & \mid & \mathrm{usubs}\ \mathit{Var\,Var\,Atom\,Atom} & \mid & \mathrm{usbb}\ \mathit{Var\,Atom\,Atom\,Atom} & \mid & \mathrm{usbbs}\ \mathit{Var\,Var\,Var\,Atom\,Atom} \\
 & \mid & \mathrm{ssub}\ \mathit{Var\,Atom\,Atom} & \mid & \mathrm{ssubs}\ \mathit{Var\,Var\,Atom\,Atom} & \mid & \mathrm{ssbb}\ \mathit{Var\,Atom\,Atom\,Atom} & \mid & \mathrm{ssbbs}\ \mathit{Var\,Var\,Var\,Atom\,Atom} \\
 & \mid & \mathrm{umul}\ \mathit{Var\,Atom\,Atom} & \mid & \mathrm{smul}\ \mathit{Var\,Atom\,Atom} & \mid & \mathrm{umull}\ \mathit{Var\,Var\,Atom\,Atom} & \mid & \mathrm{smull}\ \mathit{Var\,Var\,Atom\,Atom} \\
 & \mid & \mathrm{ushl}\ \mathit{Var\,Atom\,Num} & \mid & \mathrm{sshl}\ \mathit{Var\,Atom\,Num} & \mid & \mathrm{uspl}\ \mathit{Var\,Var\,Atom\,Num} & \mid & \mathrm{ucshl}\ \mathit{Var\,Var\,Atom\,Atom\,Num} \\
 & \mid & \mathrm{ujoin}\ \mathit{Var\,Atom\,Atom} & \mid & \mathrm{assert}\ \mathit{Apred}\ \&\&\ \mathit{RPred} & \mid & \mathrm{sspl}\ \mathit{Var\,Var\,Atom\,Num} & \mid & \mathrm{scshl}\ \mathit{Var\,Var\,Atom\,Atom\,Num} \\
 & \mid & \mathrm{sjoin}\ \mathit{Var\,Atom\,Atom} & \mid & \mathrm{assume}\ \mathit{Apred}\ \&\&\ \mathit{RPred} & & & & \\
\mathit{Decl} & ::= & \mathit{Type\,Var} & & & & \mathit{Prog} ::= \mathit{Decl}^* \ \mathit{Inst}^* & &
\end{array}
$$

Figure 13: CryptoLine syntax.

the integers, and then injecting the result back into one or more bitwords of the correct types. If the integer result cannot be represented in the output variables, then the semantics launches an explicit error. For example, the addition of two signed words with carry has two versions whose semantics are defined as follows:

$$\epsilon \xrightarrow{\mathrm{sadc}\ v\ a_0\ a_1 d} \epsilon' \qquad (v, a_0, a_1 : \sigma; d : \mathrm{bit}) \quad V = [\![a_0]\!]_\epsilon^\sigma + [\![a_1]\!]_\epsilon^\sigma + [\![d]\!]_\epsilon^{\mathrm{bit}} \quad \epsilon' = \begin{cases} \epsilon[v \mapsto V] & \text{if } \underline{\sigma} < V < \overline{\sigma} \\ \bot & \text{otherwise} \end{cases}$$

$$\epsilon \xrightarrow{\mathrm{sadcs}\ c\ v\ a_0\ a_1 d} \epsilon' \qquad (c, d : \mathrm{bit}; v, a_0, a_1 : \sigma) \quad V = [\![a_0]\!]_\epsilon^\sigma + [\![a_1]\!]_\epsilon^\sigma + [\![d]\!]_\epsilon^{\mathrm{bit}} \quad U = [\![a_0]\!]_\epsilon^\rho + [\![a_1]\!]_\epsilon^\rho + [\![d]\!]_\epsilon^{\mathrm{bit}} \quad \sigma \parallel \rho$$
$$C = \begin{cases} 0 & \text{if } U < \overline{\rho} \\ 1 & \text{otherwise} \end{cases} \qquad \epsilon' = \begin{cases} \epsilon[c, v \mapsto C, V] & \text{if } \underline{\sigma} < V < \overline{\sigma} \\ \bot & \text{otherwise} \end{cases}$$

Here, for sadc, σ denotes a signed type, a_0 and a_1 denote the input values and d denotes the input carry bit (bit is an alias for uint 1). The output value v is computed as V over the integers, and then injected back into the σ type if it falls within the range of values represented by that type ($\underline{\sigma} : \overline{\sigma}$). Otherwise, the semantics returns an error symbol $\bot$.

In the version of the instruction that returns a carry bit sadcs, then the carry is first defined using the unsigned semantics ($\sigma \parallel \rho$ fixes ρ to be the unsigned type of the same size as σ), but the final results are only consolidated in the state if the signed result of the computation V fits in σ. This stands in contrast with the unsigned version of this operator, which never returns an error.

$$\epsilon \xrightarrow{\mathrm{uadcs}\ c\ v\ a_0\ a_1 d} \epsilon[c, v \mapsto C, V] \qquad (c, d : \mathrm{bit}; v, a_0, a_1 : \rho) \quad \tilde{V} = [\![a_0]\!]_\epsilon^\rho + [\![a_1]\!]_\epsilon^\rho + [\![d]\!]_\epsilon^{\mathrm{bit}} \quad (C, V) = \begin{cases} (0, \tilde{V}) & \text{if } \tilde{V} < \overline{\rho} \\ (1, \tilde{V} - \overline{\rho}) & \text{otherwise} \end{cases}$$

The semantics of an assert statement is to leave the current state unchanged if the associated predicate evaluates to true in this state, and an error otherwise. The semantics of an assume statement is to leave the current state unchanged if the associated predicate evaluates to true in this state, and it is undefined behavior otherwise (for simplicity, one can think of assuming false as an instruction that blocks the execution of the program). Note that in the CryptoLine semantics, the evaluation of algebraic predicates is carried out over the integers: the semantics first computes the interpretations of the values of atoms occurring in a predicate, and then evaluates the predicate. For range predicates, the predicate validity is checked directly over bit-vectors.

Verification of CryptoLine programs. CryptoLine is geared towards automatic verification using two families of tools: SMT solvers supporting the bit vector theory SMT-LIB2, and Computer Algebra Systems (CAS). The CryptoLine tool can convert a partial correctness Hoare triple proof goal for a CryptoLine program into a set of sufficient proof obligations. Note that partial correctness here means that the post-condition holds whenever the evaluation of the program reached a final state. Because the program is straight-line and verified to be safe, the only possible cause for not reaching a final state is blocking due to the attempt to evaluate an assume statement which turns out to be false. The contract (pre- and post- conditions) are given as special (syntactically distinguished) instances of assert instructions over the program inputs and program final state. Proof obligations associated with range predicates are discharged using SMT solvers. Obligations associated with algebraic predicates are discharged using CAS.

The safety of a CryptoLine program is proved using a dedicated verification condition generator that checks for the property that the CryptoLine program does not terminate in an error state.

B Specifications of the ML-KEM NTT

Figure 14 is presented here due to space constraints.

C Extension to vectorized instructions

The reference implementation of the NTT function that we discussed in Section 8 doesn't fully show the advantages of the automation offered by CryptoLine wrt a fully interactive proof in EasyCrypt. This is because, as we mentioned above, the specification and implementation are close enough to allow an interactive proof with reasonable effort. This is *not* the case for the AVX2 implementation in Figure 16. Here, there is no discernable relation between the control flow of the specification and the implementation, so being able to establish automatically that the implementation computes the algebraic specification indeed provides a huge benefit wrt to the overall proof effort required for an interactive proof. However, the use of vectorized instructions in this implementation requires that we enrich our interaction with CryptoLine to permit reasoning about vectorized instructions. We first briefly explain the support CryptoLine provides at this level.

Vectorized instructions in CryptoLine. CryptoLine does not have support for architecture-specific vectorized extensions such as AVX2. Instead, CryptoLine offers a generic mechanism to describe programs that rely on vectorized operations. This mechanism, in

Algorithm 9 NTT(f)

Computes the NTT representation $\hat{f}$ of the given polynomial $f \in R_q$.

Input: array $f \in \mathbb{Z}_q^{256}$. ▷ the coefficients of the input polynomial

Output: array $\hat{f} \in \mathbb{Z}_q^{256}$. ▷ the coefficients of the NTT of the input polynomial

```
1:  $\hat{f} \leftarrow f$                                      ▷ will compute in place on a copy of input array
2:  $i \leftarrow 1$
3:  for ($len \leftarrow 128$; $len \geq 2$; $len \leftarrow len/2$)
4:      for ($start \leftarrow 0$; $start < 256$; $start \leftarrow start + 2 \cdot len$)
5:          $zeta \leftarrow \zeta^{\mathsf{BitRev}_7(i)} \bmod q$
6:          $i \leftarrow i+1$
7:          for ($j \leftarrow start$; $j < start + len$; $j$++)
8:              $t \leftarrow zeta \cdot \hat{f}[j+len]$                ▷ steps 8-10 done modulo $q$
9:              $\hat{f}[j+len] \leftarrow \hat{f}[j] - t$
10:             $\hat{f}[j] \leftarrow \hat{f}[j] + t$
11:         end for
12:     end for
13: end for
14: return $\hat{f}$
```

```
fn _poly_ntt(reg ptr u16[MLKEM_N] rp) →
  reg ptr u16[MLKEM_N] {
 zetasp = jzetas; zetasctr = 0; len = 128;
 while (len >= 2) {
  start = 0;
  while (start < 256) {
   zetasctr += 1; zeta = zetasp[(int)zetasctr];
   j = start; cmp = start; cmp += len;
   while (j < cmp) {
    s = rp[(int)j]; m = s; offset = j; offset += len;
    t = rp[(int)offset]; t = __fqmul(t, zeta); m -= t; t += s;
    rp[(int)offset] = m; rp[(int)j] = t; j += 1;
   } start = j; start += len;
  } len >>= 1; }
 rp = __poly_reduce(rp); return rp; }
```

Figure 14: NTT as specified in FIPS203 (left) and Jasmin reference implementation (right).

very short terms, works as follows. Program variables can be typed as vectors of scalar variables, e.g., one can initialize a variable $\%v$, whereby the percentage symbol denotes a vector, with a list of scalar variables $[v_1, \ldots, v_n]$, where all variables in the vector must be of the same CryptoLine type. Conversely, a vector $\%v$ can be assigned to a list of scalar variables of the appropriate type, or it can be *cast* into a list of variables of different types, conditionally on the natural size restrictions being satisfied. Furthermore, CryptoLine offers the natural lift of scalar operations to vectors by interpreting their application pointwise. The vectorized instruction is simply signalled with a % sign. Operations that break the trivial parallel computation pattern can be emulated by *unserializing* a vector into a set of scalar variables and performing computations over the scalar variables using the standard CryptoLine approach. For example, a permutation can be modelled by assigning the vector to scalar variables and initializing a new vector with the permuted scalars.

The problem. A simplified Jasmin program with vector instructions does not necessarily have a natural pretty-printed CryptoLine counterpart. The problem is that a CryptoLine vector variable has a fixed type for its contents, whereas a long register in Jasmin can be used as an input to instructions that will handle its contents with different vector types. One possible solution to this problem would be to take the (certified) simplification of the Jasmin program one step further and convert it into a scalar program over which we could apply the approach we described in the previous examples. This is a route we leave to explore in future work. In this paper, we give a simple solution that requires placing a slightly more trust in the pretty printing routine.

Our solution. We enrich our translation to CryptoLine with knowledge on how how each AVX2 instruction interprets the vector register as a list of scalar values of specific size (e.g VPADD_16u16 treats a 256-bit register as a vector of 16x16-bit values). This means that we can guarantee the generation of a well-typed CryptoLine program as follows. We maintain the invariant that all variables of a long word type (i.e. greater than the machine word size) are initialized in CryptoLine as vector variables on their first assignment, i.e. their type is defined by the operation that first assigns to them.[11] Then, each AVX2 instruction is pretty printed as follows: for each vector input argument, it uses the cast operation in CryptoLine to create a list of scalar variables representing the input. The types of these scalar variables are determined by the AVX2 instruction that will be computed. These scalar variables are then packed into a new (fresh variable) vector input that can be passed to the vectorized CryptoLine instruction. In this way, even if adjacent AVX2 instructions are processing long registers with different vector granularity, the output CryptoLine program will be valid. The disadvantage of this approach is that the resulting program may certainly include unnecessary cast operations: this will happen whenever two adjacent vector instructions are operating over the same sub-word size (e.g. VPADD_16u16 followed by VPSUB_16u16). In order to reduce the complexity of the model, we introduced a non-verified simplification in our pretty-printing routine that identifies and removes redundant casts. This simple heuristic works well for the AVX2 implementation of the NTT, which is very well behaved wrt its use of SIMD operations—all vectorized instructions work over 256-bit words seen as 16x16-bit sub-words, so essentially all casts are removed.

A final extension to the pretty-printing mechanism was necessary to allow writing Jasmin annotations and contracts that refer to sub-words of long inputs and outputs. To this end, we allow the Jazzline user to annotate the types of long inputs and outputs to functions with their corresponding vector type and introduce abstract predicates that permit referring to subwords in the syntax of both range and algebraic predicates. This meta-information informs the pretty-printing mechanism of how long inputs and outputs to a Jasmin function should be partitioned into scalar variables: the resulting CryptoLine program takes scalar variables as inputs and outputs, so its contract is written in a purely scalar form. The first/last instructions in the pretty-printed program perform the necessary packing/unpacking to guarantee consistency. Similarly, assert and assume annotations are expressed over scalar variables resulting from unpacking instructions.

[11]The exception here are input and output variables, which we explain later.

The proof. To demonstrate the feasiblity of this extension we targeted the NTT implementation of ML-KEM-768. The approach here largely follows the one described for the reference implementation of the same function, the main difference being the way the pre- and post-conditions are specified. As stated previously, the use of input polynomials as arrays of vector registers instead of arrays of scalar values, requires us to use an unpacking predicate (in this case u256_as_16u16) and nested big operators in order to index each coefficient individually. The contract annotated in the Jasmin function is shown in Figure 15. (Note the use of rpi as a phantom input that allows simplifying the writing of the NTT algebraic formula.)

The use of EasyCrypt in the proof of the AVX2 function is very close to what we described for the scalar version of the algorithm. Matching the algebraic contract resulting from the CryptoLine proof to the intended EasyCrypt contract follows by applying the same lemma relating the algebraic definition of the NTT to its imperative specification. We also factored out the proof of the Barrett reduction step into a proof goal that was assumed in CryptoLine and discharged interactively in EasyCrypt. Moreover, we did the same for some subroutines that are performing sequences of permutations in order to reshuffle intermediate values.

The take-away message from this example is that our approach is conceptually general enough to allow reusing CryptoLine proofs relying on vectorized instructions for similar Jasmin programs. However, we have also learned that extending the framework to support the wide variety of SIMD instructions included in various architectures will give rise to a complex pretty-printer. As mentioned above, to avoid that, we plan to explore certified compilation steps that generically convert vectorized programs into equivalent scalar ones.

```
fn _poly_ntt(reg ptr u16[MLKEM_N] rpi, #[vect = "u16x16"] reg ptr u256[MLKEM_N/16] rp, #[vect = "u16x16"] reg u256 qx16 vx16)  → #[vect = "u16x16"]  reg ptr u256[MLKEM_N/16]
  requires #[prover=smt] {(\all (ii ∈ 0:MLKEM_N/16) (\all (ij ∈ 0:16) ((−2)∗MLKEM_Q ≤16s u256_as_16u16(rp[ii],ij) && u256_as_16u16(rp[ii],ij) <16s 2∗MLKEM_Q)))}
  requires #[prover=smt] {(\all (ij ∈ 0:16) (u256_as_16u16(qx16, ij) == MLKEM_Q && u256_as_16u16(vx16, ij) == 20159))}
  requires #[prover=cas] {(\all (ij ∈ 0:16) (u256_as_16u16(qx16, ij) == MLKEM_Q && u256_as_16u16(vx16, ij) == 20159))}
  requires #[prover=smt] {(\all (ii ∈ 0:MLKEM_N/16) (\all (ij ∈ 0:16) (u256_as_16u16(rp[ii],ij) == rpi[16∗ii+ij])))}
  ensures #[prover=smt] {(\all (ii ∈ 0:MLKEM_N/16) (\all (ij ∈ 0:16) (MLKEM_Q ≤16s u256_as_16u16(result.0[ii],ij) && u256_as_16u16(result.0[ii],ij) <16s MLKEM_Q)))}
  ensures #[prover=cas] {(\all (ii ∈ 0:MLKEM_N/16) (\all (ij ∈ 0:16) (u256_as_16u16(rp[ii],ij) == rpi[16∗ii+ij]))) &&
   \all (pp ∈ 0:MLKEM_N/16) (\all (pj ∈ 0:8) ((eqmod(\big [+/mon0(0)] (ii ∈ 0:MLKEM_N) (mon(0,ii,u16i(rpi[ii]))),
      mon0(u16i(u256_as_16u16(result.0[pp],2∗pj))) + mon(0,1,u16i(u256_as_16u16(result.0[pp],2∗pj+1))), pair(mon0(MLKEM_Q),mon(0,2,1) − mon0(u16i(zetas_power[pp])))))))}
```

Figure 15: Contract of the Jazzline proof for the AVX2 NTT.

```
inline fn __butterfly64x(reg u256 rl0 rl1 rl2 rl3 rh0 rh1 rh2
  rh3 zl0 zl1 zh0 zh1 qx16) → reg u256, reg u256, reg u256,
  reg u256, reg u256, reg u256, reg u256, reg u256 {
 t0 = #VPMULL_16u16(zl0, rh0); t1 = #VPMULH_16u16(zh0, rh0);
 t2 = #VPMULL_16u16(zl0, rh1); t3 = #VPMULH_16u16(zh0, rh1);
 t4 = #VPMULL_16u16(zl1, rh2); t5 = #VPMULH_16u16(zh1, rh2);
 t6 = #VPMULL_16u16(zl1, rh3); t7 = #VPMULH_16u16(zh1, rh3);
 t0 = #VPMULH_16u16(t0, qx16); t2 = #VPMULH_16u16(t2, qx16);
 t4 = #VPMULH_16u16(t4, qx16); t6 = #VPMULH_16u16(t6, qx16);
 rh1 = #VPSUB_16u16(rl1, t3); rl1 = #VPADD_16u16(t3, rl1);
 rh0 = #VPSUB_16u16(rl0, t1); rl0 = #VPADD_16u16(t1, rl0);
 rh3 = #VPSUB_16u16(rl3, t7); rl3 = #VPADD_16u16(t7, rl3);
 rh2 = #VPSUB_16u16(rl2, t5); rl2 = #VPADD_16u16(t5, rl2);
 rh0 = #VPADD_16u16(t0, rh0); rl0 = #VPSUB_16u16(rl0, t0);
 rh1 = #VPADD_16u16(t2, rh1); rl1 = #VPSUB_16u16(rl1, t2);
 rh2 = #VPADD_16u16(t4, rh2); rl2 = #VPSUB_16u16(rl2, t4);
 rh3 = #VPADD_16u16(t6, rh3); rl3 = #VPSUB_16u16(rl3, t6);
 return rl0, rl1, rl2, rl3, rh0, rh1, rh2, rh3; }

fn _poly_ntt(reg ptr u16[MLKEM_N] rpi, reg ptr u256[MLKEM_N/16] rp,
     reg u256 qx16 vx16) → reg ptr u256[MLKEM_N/16] {
 j = 0; zeta0 = #VPBROADCAST_8u32(jzetas_exp.[u32 j]);
 j = 4; zeta1 = #VPBROADCAST_8u32(jzetas_exp.[u32 j]);
 r0 = rp[0 ]; r1 = rp[1 ]; r2 = rp[2 ]; r3 = rp[3 ];
 r4 = rp[8 ]; r5 = rp[9 ]; r6 = rp[10]; r7 = rp[11];
 r0, r1, r2, r3, r4, r5, r6, r7 =
  __butterfly64x(r0, r1, r2, r3, r4, r5, r6, r7,
                    zeta0, zeta0, zeta1, zeta1, qx16);
 rp[0 ] = r0; rp[1 ] = r1; rp[2 ] = r2; rp[3 ] = r3;
 rp[8 ] = r4; rp[9 ] = r5; rp[10] = r6; rp[11] = r7;
 r0 = rp[4 ]; r1 = rp[5 ]; r2 = rp[6 ]; r3 = rp[7 ];
 r4 = rp[12]; r5 = rp[13]; r6 = rp[14]; r7 = rp[15];
 r0, r1, r2, r3, r4, r5, r6, r7 =
  __butterfly64x(r0, r1, r2, r3, r4, r5, r6, r7,
                    zeta0, zeta0, zeta1, zeta1, qx16);
 rp[12] = r4; rp[13] = r5; rp[14] = r6; rp[15] = r7;
 for i=0 to 2 {
  j = 8 + 392∗i; zeta0 = #VPBROADCAST_8u32(jzetas_exp.[u32 j]);
  j = 12 + 392∗i; zeta1 = #VPBROADCAST_8u32(jzetas_exp.[u32 j]);
  if ( i == 0) { r4 = r0; r5 = r1; r6 = r2; r7 = r3; }
  else { r4 = rp[4+8∗i]; r5 = rp[5+8∗i]; r6 = rp[6+8∗i]; r7 = rp[7+8∗i]; }
  r0 = rp[0+8∗i]; r1 = rp[1+8∗i]; r2 = rp[2+8∗i]; r3 = rp[3+8∗i];
  r0, r1, r2, r3, r4, r5, r6, r7 =
   __butterfly64x(r0, r1, r2, r3, r4, r5, r6, r7,
                  zeta0, zeta0, zeta1, zeta1, qx16);
  j = 16 + 392∗i; zeta0 = jzetas_exp.[u256 j];
  j = 48 + 392∗i; zeta1 = jzetas_exp.[u256 j];
  r0, r4 = _shuffle8(r0, r4); r1, r5 = _shuffle8(r1, r5);
  r2, r6 = _shuffle8(r2, r6); r3, r7 = _shuffle8(r3, r7);
  r0, r4, r1, r5, r2, r6, r3, r7 =
   __butterfly64x(r0, r4, r1, r5, r2, r6, r3, r7,
                  zeta0, zeta0, zeta1, zeta1, qx16);
  j = 80 + 392∗i; zeta0 = jzetas_exp.[u256 j];
  j = 112 + 392∗i; zeta1 = jzetas_exp.[u256 j];
  r0, r2 = _shuffle4(r0, r2); r4, r6 = _shuffle4(r4, r6);
  r1, r3 = _shuffle4(r1, r3); r5, r7 = _shuffle4(r5, r7);
  r0, r2, r4, r6, r1, r3, r5, r7 =
   __butterfly64x(r0, r2, r4, r6, r1, r3, r5, r7,
                  zeta0, zeta0, zeta1, zeta1, qx16);
  j = 144 + 392∗i; zeta0 = jzetas_exp.[u256 j];
  j = 176 + 392∗i; zeta1 = jzetas_exp.[u256 j];
  r0, r1 = _shuffle2(r0, r1); r2, r3 = _shuffle2(r2, r3);
  r4, r5 = _shuffle2(r4, r5); r6, r7 = _shuffle2(r6, r7);
  r0, r1, r2, r3, r4, r5, r6, r7 =
   __butterfly64x(r0, r1, r2, r3, r4, r5, r6, r7,
                  zeta0, zeta0, zeta1, zeta1, qx16);
  j = 208 + 392∗i; zeta0 = jzetas_exp.[u256 j];
  j = 240 + 392∗i; zeta1 = jzetas_exp.[u256 j];
  r0, r4 = _shuffle1(r0, r4); r1, r5 = _shuffle1(r1, r5);
  r2, r6 = _shuffle1(r2, r6); r3, r7 = _shuffle1(r3, r7);
  r0, r4, r1, r5, r2, r6, r3, r7 =
   __butterfly64x(r0, r4, r1, r5, r2, r6, r3, r7,
                  zeta0, zeta0, zeta1, zeta1, qx16);
  j = 272 + 392∗i; zeta0 = jzetas_exp.[u256 j];
  j = 304 + 392∗i; zeta2 = jzetas_exp.[u256 j];
  j = 336 + 392∗i; zeta1 = jzetas_exp.[u256 j];
  j = 368 + 392∗i; zeta3 = jzetas_exp.[u256 j];
  r0, r4, r2, r6, r1, r5, r3, r7 =
   __butterfly64x(r0, r4, r2, r6, r1, r5, r3, r7,
                  zeta0, zeta1, zeta2, zeta3, qx16);
  r0 = __red16x(r0, qx16, vx16); r4 = __red16x(r4, qx16, vx16);
  r2 = __red16x(r2, qx16, vx16); r6 = __red16x(r6, qx16, vx16);
  r1 = __red16x(r1, qx16, vx16); r5 = __red16x(r5, qx16, vx16);
  r3 = __red16x(r3, qx16, vx16); r7 = __red16x(r7, qx16, vx16);
  rp[0+8∗i] = r0; rp[1+8∗i] = r4; rp[2+8∗i] = r1; rp[3+8∗i] = r5;
  rp[4+8∗i] = r2; rp[5+8∗i] = r6; rp[6+8∗i] = r3; rp[7+8∗i] = r7;
 }

 return rp;
}
```

Figure 16: Snippets of the Jasmin AVX2 implementation.