Faster Verification of Faster Implementations: Combining Deductive and Circuit-Based Reasoning in EasyCrypt

José Bacelar Almeida*, Gustavo Xavier Delerue Marinho Alves†‡, Manuel Barbosa†§, Gilles Barthe§¶, Luís Esquível†, Vincent Hwang§, Tiago Oliveira‖, Hugo Pacheco†, Peter Schwabe§ and Pierre-Yves Strub‡
*Universidade do Minho and INESC TEC
jba@di.uminho.pt
†Universidade do Porto (FCUP) and INESC TEC
gxdelerue@gmail.com, mbb@fc.up.pt, hpacheco@fc.up.pt, luis.esquivel.costa@gmail.com
‡PQShield
pierre-yves.strub@pqshield.com
§Max Planck Institute for Security and Privacy
gilles.barthe@mpi-sp.org, vincentvbh7@gmail.com, peter@cryptojedi.org
¶IMDEA Software Institute
‖SandboxAQ
tiago.oliveira@sandboxquantum.com
**Radboud University

Abstract—**We propose a hybrid formal verification approach that combines high-level deductive reasoning and circuit-based reasoning and apply it to highly optimized cryptographic assembly code. Our approach permits scaling up formal verification in two complementary directions: 1) it reduces the proof effort required for low-level functions where the computation logics are obfuscated by the intricate use of architecture-specific instructions and 2) it permits amortizing the effort of proving one implementation by using equivalence checking to propagate the guarantees to other implementations of the same computation using different optimizations or targeting different architectures. We demonstrate our approach via an extension to the EasyCrypt proof assistant and by revisiting formally verified implementations of ML-KEM in Jasmin. As a result, we obtain the first formally verified implementation of ML-KEM that offers performance comparable to the fastest non-verified implementation in x86-64 architectures.**

1. Introduction

The deployment of cryptographic implementations requires extremely efficient architecture-specific code. As a consequence, cryptographic libraries typically rely on highly optimized assembly implementations, making platform-specific use of data layout, control flow and ISA extensions. In fact, for a given low-level function, multiple implementations must often coexist. Developing and maintaining this code is a difficult and error-prone process. It requires deep domain-specific expertise, not only on the functionality being implemented and the target architecture, but also on implementation-level techniques to mitigate the risks of timing and other microarchitectural attacks. For these reasons, cryptography libraries increasingly turn to high-assurance methods to prove that these implementations are correct.

One popular approach to prove the correctness optimized implementations is to show their equivalence with reference implementations. This approach has been applied successfully in the cryptographic context, with automated tools such as Axe [1] and CryptoLec [2]. Unfortunately, automated equivalence checking of highly optimized implementations of complex algorithms such as those occurring in post-quantum cryptography remains computationally intractable.

An alternative approach is to use deductive program verification, either with standalone tools, or via embeddings into proof assistants. This approach is used extensively in the context of high-assurance cryptography [3], and blends well with complementary verification tasks required for cryptographic constructions, namely security proofs. However, the manual effort of deductive program verification for optimized cryptographic implementations is enormous and, for this reason, this approach does not scale to permit covering the many hand-optimized assembly routines that occur in real-world cryptography libraries.

Approach. We present a program verification framework that supports fine-grained interactions between equivalence checking and deductive program verification. The main high-level idea of our approach is to introduce a hybrid representation of programs, where selected program fragments can be written as circuits. Critically, our representation of circuits is architecture-independent, so that sophisticated data layouts, such as words of non-standard size, have a direct translation into our hybrid representation. An additional benefit of the circuit representation of programs is that it allows exhibiting the parallel structure of computations. This makes it possible to reason efficiently about maps, permutations, and other simple parallel computation patterns that are often (conceptually) present in highly optimized data-wrangling functions. The other high-level idea of our approach is that the conversion from programs to circuits is driven by and freely interleaved with deductive verification. This makes it possible to apply equivalence checking both compositionally and iteratively. In particular, our approach allows users to decompose programs into a hybrid form where circuit-based reasoning permits simplifying some components and recover a logical structure that is hidden by architectural aspects. The simplified program can then be, again, analysed deductively or using circuit-based reasoning.

A main application of our approach are data-wrangling functions of the kind discussed above. While these functions perform conceptually simple computations, they represent a significant burden for verification, for two reasons: (1) they are particularly challenging to prove, due to aggressive architecture-specific optimizations; (2) the cost of their verification by purely deductive methods is not amortized across different architectures. In contrast, our approach makes verification of these functions nearly automated, and allows to reuse most of the proofs across architectures. Moreover, the resulting proofs are robust to small modifications of implementations that are introduced over time to further improve performance. This massively reduces the effort for proof maintenance through the whole lifecycle of high-assurance cryptographic software.

We implement our approach in the EasyCrypt proof assistant [4], [5]. For concreteness, we focus on cryptographic implementations written in the Jasmin language [6], [7]. We evaluate our approach on optimized Jasmin implementations of the lattice-based key-encapsulation mechanism ML-KEM, which is based on Kyber [8], [9], and which has recently been standardized by NIST as FIPS 203 [10]. One of our main examples is the implementation of a rejection sampling routine in ML-KEM, which is optimized using AVX2 vector instructions. Verifying the correctness of this optimized routine was left as an open challenge by prior works [11], [12]. We also demonstrate applicability to a critical component in an optimized Jasmin implementation of the SHA-3 standard [13].

Summary of contributions. We propose an approach to the formal verification of low-level cryptographic code that combines deductive verification and circuit-based reasoning. Our approach is conceptually simple and builds on the theoretical foundations that underly EasyCrypt, leveraging the ability to decompose complex functional-correctness proof goals expressed as (relational) Hoare judgements into simpler goals that can be soundly restated as equivalences between circuits. Crucially, by carefully selecting the components that are interpreted as circuits, often after applying simple high-level transformations that greatly simplify them, we can perform equivalence checking in a scalable way.

Our main contributions are as follows:

- We extend EasyCrypt with a new backend that permits circuit-based reasoning and new features that allow simplifying programs and converting them into circuit form.
- We create a general framework for connecting the axiomatic semantics for languages formalized in EasyCrypt to the circuit-based reasoning backend. Our instantiation supports programs in Jasmin for x86-64 (including AVX2 vector extensions) and ARMv7-M architectures.
- We demonstrate the advantages of our approach by presenting a new formally verified implementation of ML-KEM that offers a performance that is comparable to that of the fastest non-verified implementations available in x86-64 plaforms supporting AVX2. The hybrid reasoning now available in EasyCrypt was instrumental to formally verify the optimized rejection sampling that prior works left as an open challenge.
- Additionally, we demonstrate proof-engineering advantages of our method, by focusing on a compression routine also present in ML-KEM implementations, which requires working with non-standard word sizes (10-bit words). We show that several versions of this routine, for x86-64 and ARM architectures can be proved correct with a fraction of the effort that was reported for the original deductive verification proof in EasyCrypt [11], [12]. Moreover, one can easily amortize proof effort by transferring functional correctness results from one implementation to another.
- Finally, to illustrate the usage of our techniques in the symmetric cryptography setting, we report our results on verifying the AVX2 implementation of the Keccak permutation used in the SHA-3 standard.

Our development is publicly available via the EasyCrypt repository,[1] and the examples from the Formosa Crypto organization.[2]

Paper structure. In the next section we introduce our motivating example. In Section 3 we provide some background on Jasmin, EasyCrypt and the formal verification methodology that combines these tools. In Sections 4 and 5 we describe how our methodology works in EasyCrypt, first explaining how circuit-based reasoning can be used to discharge (relational) Hoare triples, and then discussing how hybrid reasoning scales better than deductive and circuit-based reasoning independently. In Section 6 we describe engineering effort invested in this new version of EasyCrypt, and in Section 7 we describe the use cases that support our results.

2. The Challenge: AVX2 Rejection Sampling

As motivating example we will use the Jasmin function presented in Figure 1. This function sits at the core of the op-

[1] https://github.com/EasyCrypt/easycrypt
[2] https://github.com/formosa-crypto/

```
inline fn __gen_matrix_buf_rejection_filter48
( reg mut ptr u16[MLKEM_N] pol, reg u64 counter, reg const ptr u8[BUF_size] buf, reg u64
    buf_offset, reg u256 load_shuffle, reg u256 mask, reg u256 bounds, reg ptr u8[2048] sst,
    reg u256 ones) → reg ptr u16[MLKEM_N], reg u64 {
  reg u256 f0 f1 g0 g1, shuffle_0 shuffle_1 shuffle_t;
  reg u128 shuffle_0_1 shuffle_1_1;
  reg u64 good t0_0 t0_1 t1_0 t1_1;
  f0 = #VPERMQ(buf.[u256 (int) buf_offset + 0 ], (4u2)[2,1,1,0]);
  f1 = #VPERMQ(buf.[u256 (int) buf_offset + 24], (4u2)[2,1,1,0]);
  f0 = #VPSHUFB_256(f0, load_shuffle); f1 = #VPSHUFB_256(f1, load_shuffle);
  g0 = #VPSRL_16u16(f0, 4); g1 = #VPSRL_16u16(f1, 4);
  f0 = #VPBLEND_16u16(f0, g0, 0xAA); f1 = #VPBLEND_16u16(f1, g1, 0xAA);
  f0 = #VPAND_256(f0, mask); f1 = #VPAND_256(f1, mask);
  g0 = #VPCMPGT_16u16(bounds, f0); g1 = #VPCMPGT_16u16(bounds, f1);
  g0 = #VPACKSS_16u16(g0, g1); good = #VPMOVMSKB_u256u64(g0);
  t0_0 = good; t0_0 &= 0xFF;
  shuffle_0 = (256u) #VMOV(sst[u64 (int)t0_0]);
  ?{}, t0_0 = #POPCNT_64(t0_0); t0_0 += counter;
  t0_1 = good; t0_1 >>= 16; t0_1 &= 0xFF;
  shuffle_0_1 = #VMOV(sst[u64 (int)t0_1]);
  ?{}, t0_1 = #POPCNT_64(t0_1); t0_1 += t0_0;
  t1_0 = good; t1_0 >>= 8; t1_0 &= 0xFF;
  shuffle_1 = (256u) #VMOV(sst[u64 (int)t1_0]);
  ?{}, t1_0 = #POPCNT_64(t1_0); t1_0 += t0_1;
  t1_1 = good; t1_1 >>= 24; t1_1 &= 0xFF;
  shuffle_1_1 = #VMOV(sst[u64 (int)t1_1]);
  ?{}, t1_1 = #POPCNT_64(t1_1); t1_1 += t1_0;
  shuffle_0 = #VINSERTI128(shuffle_0, shuffle_0_1, 1);
  shuffle_1 = #VINSERTI128(shuffle_1, shuffle_1_1, 1);
  shuffle_t = #VPADD_32u8(shuffle_0, ones);
  shuffle_0 = #VPUNPCKL_32u8(shuffle_0, shuffle_t);
  shuffle_t = #VPADD_32u8(shuffle_1, ones);
  shuffle_1 = #VPUNPCKL_32u8(shuffle_1, shuffle_t);
  f0 = #VPSHUFB_256(f0, shuffle_0); f1 = #VPSHUFB_256(f1, shuffle_1);
  pol.[u128 2*counter] = (128u)f0; pol.[u128 2*t0_0] = #VEXTRACTI128(f0, 1);
  pol.[u128 2*t0_1] = (128u)f1; pol.[u128 2*t1_0] = #VEXTRACTI128(f1, 1);
  counter = t1_1;
  return pol, counter;
}
```

Load 64 bytes from 56 positions

Arrange as 2 groups of 16x12-bit values (each stored as u256=16u16)

Compute 32 bound-checks

For each group of 8 check bits, count number of accepted words and use them to define offsets for storage. For each group of 8 check bits, define a permutation pattern for storage of 0 to 8 accepted words.

Prepare permutation descriptors

Permute and store

Figure 1. Core rejection sampling routine.

timized implementations of ML-KEM[3] and it provides non-trivial performance gains in a computationally expensive step in the construction: the generation of a *random-looking* matrix in $R_q^{k\times k}$. Here, k is a small integer in $\{2, 3, 4\}$ and R_q denotes the polynomial ring $\mathbb{Z}_q[X]/(X^{256}+1)$ for $q = 3329$.

The goal of the function is to convert a sequence of 48 bytes coming from the output of an eXtendable Output Function (XOF), in this case SHAKE128, into 32 candidates for polynomial coefficients represented as 16-bit words in the range $[0, 2^{12})$, and then copying those that fall in the range $[0, q)$ to an output buffer. This AVX2-specific optimization was first proposed by Gueron and Schlieker [14] for optimizing the lattice-based KEM NewHope [15]. It was then adapted by the Kyber team in the submission to the NIST PQC competition [9], which eventually gave rise to the ML-KEM standard [10].

Due to its impact on performance, which we demonstrate in Section 7, this routine and adaptations to other SIMD-enabled architectures will be deployed in widely used software implementations. However, as we believe will become apparent in this section, guaranteeing that such implementations are correct is far from trivial.

What the code is doing. The high-level, intuitive, specification of the computation carried out by this function is given by the following EasyCrypt expression, where bs2int denotes converting a bit string to integers, chunk 12 breaks down a long bit string into chunks of size 12, and the remaining operators perform standard list and array operations.

```
op filter48(buf : W8.t Array536.t, pol : W16.t Array256.t,
            counter : int, buf_ofset : int) =
    let buf_unused = take 48 (drop buf_offset (to_list buf))
  in let candidates = map bs2int (chunk 12 (Bytes2Bits bufunused))
  in let good = filter (fun x ⇒  x < q) candidates
  in let ngood = size good
  in let pol = Array256.init (fun i ⇒  if count ≤ i ∧ i < count + ngood
                                        then good[i – ngood]
                                        else pol[i])
  in (pol, count + ngood)
```

There are two data inputs: 1) buf is an array of 536 bytes, containing fresh (pseudo-)randomness starting at position buf_offset and 2) pol is a polynomial, represented as an array of 16-bit words of size 256, which has already been partially filled with count coefficients. The function processes

[3] Our formally verified implementation of ML-KEM is proved correct wrt the same abstract specification that was proved secure in [12], and therefore retains the security guarantees that are discussed for the implementations in that work. In terms of implementation correctness, and because the differences to the original Kyber specification in [11] are very small, the results could be easily extended to cover Kyber as well.

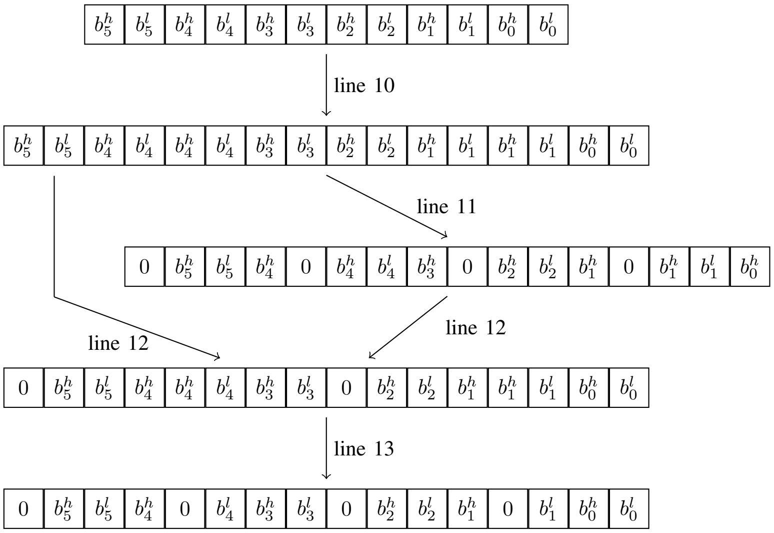
b^h_5 b^l_5 b^h_4 b^l_4 b^h_3 b^l_3 b^h_2 b^l_2 b^h_1 b^l_1 b^h_0 b^l_0
line 10
b^h_5 b^l_5 b^h_4 b^l_4 b^h_4 b^l_4 b^h_3 b^l_3 b^h_2 b^l_2 b^h_1 b^l_1 b^h_1 b^l_1 b^h_0 b^l_0
line 11
0 b^h_5 b^l_5 b^h_4 0 b^h_4 b^l_4 b^h_3 0 b^h_2 b^l_2 b^h_1 0 b^h_1 b^l_1 b^h_0
line 12
line 12
0 b^h_5 b^l_5 b^h_4 b^h_4 b^l_4 b^h_3 b^l_3 0 b^h_2 b^l_2 b^h_1 b^h_1 b^l_1 b^h_0 b^l_0
line 13
0 b^h_5 b^l_5 b^h_4 0 b^l_4 b^h_3 b^l_3 0 b^h_2 b^l_2 b^h_1 0 b^l_1 b^h_0 b^l_0

Figure 2. Rearranging 6 bytes read from memory as four 12-bit words. This pattern occurs 4 times across each of the 256-bit registers f0 and f1, so that in the end there are sixteen 12-bit words in each of them.

48 randomness bytes, so the caller must ensure that:[4] i. buf_offset $+ 48 + 8 \leq 536$; and ii. $48 * 8/12 = 32$ can be potentially stored in pol i.e. counter $+ 32 \leq 256$. Then the function takes the next 48 bytes, which it chunks into 12-bit integers (32 of them) and stores the ones that are in the range $[0..q)$ in the next available positions in pol.

The Jasmin code uses AVX2 intrinsics heavily. We added comments to highlight the following computation steps:

1) **Lines 6-7:** Data is read from memory. Two loads are performed into two 256-bit registers f0 and f1. Each 32-byte load, seen as four 64-bit words $ABCD$ actually already permutes the input into $ABBC$ ignoring the last 8 bytes. This justifies the overlap in the two loads.
2) **Lines 8-11:** The 24 bytes loaded into f0 and f1 are expanded into sixteen 12-bit numbers. This process is depicted in Figure 2: after a permutation that realigns the 4-bit subwords, auxilliary variables g0 and g1 are used to store right-shifted versions of f0 and f1. This means that half of the required 12-bit values are now in g0 and g1, whereas the remaining ones are in f0 and f1 but they need a masking step. The process is concluded by blending pairwise g0 and f0 (resp. g1 and f1) and masking the results to ensure that the 4 most significant bits are 0.
3) **Lines 12-13:** All 12-bit values are then checked to be within the $[0, q)$ bound, resulting in 32-boolean bound-check results. This ends the first part of the function, where the parallel computation is carried out using two 256-bit tracks, each handling 16 candidate values. From here onwards, the computation is carried out in four tracks, each taking care of 8 candidate values, as follows.
4) **Lines 14-25:** For each track, the 8 candidate values need to be permuted, placing the good values on the left, while preserving their order. Since there are 8 bound-check bits for these candidates, this means that there are 256 possible reorderings that may occur. The next step in the computation looks at the bound-check bits, recovers from a table in memory (sst) the permutation pattern that will rearrange them correctly, and computes how many valid candidates were found. The function computes four permutation patterns, one for each track, but note that the values are still stored in 256-bit registers. So, logically, each 256-bit register is seen as a pair 128-bit words, each storing eight 16-bit values.
5) **Lines 26-31:** To conclude the computation, the code then transforms the permutation patterns into sub-word indexing information: variables shuffle_0 and shuffle_1 are filled with the byte-level index pattern that permutes the bytes in f0 and f1 in the required way. At the low level, first the indices of the least significant bytes of each 16-bit word are stored in shuffle_0 and shuffle_1, and then these are used to compute the indices of the most significant bytes incrementing by one. The two sets of indices are then merged into the final values of shuffle_0 and shuffle_1.
6) **Lines 32-34:** Finally, the candidates in f0 and f1 are permuted, creating four sets of values that need to be copied into the output polynomial. The function concludes by making these copies in order, so that the rejected candidates from earlier tracks are overwritten by accepted candidates in subsequent tracks.

Proving functional correctness. A formal proof that the function in Figure 1 correctly implements the high-level specification poses significant challenges:

1) The logical structure of the computation done by the

[4] The extra 8-byte slack in the admissible range of buf_offset comes from the fact that the function actually reads $24 + 32 = 56$ bytes from buf, as can be seen in lines 7 and 8 of the listing.

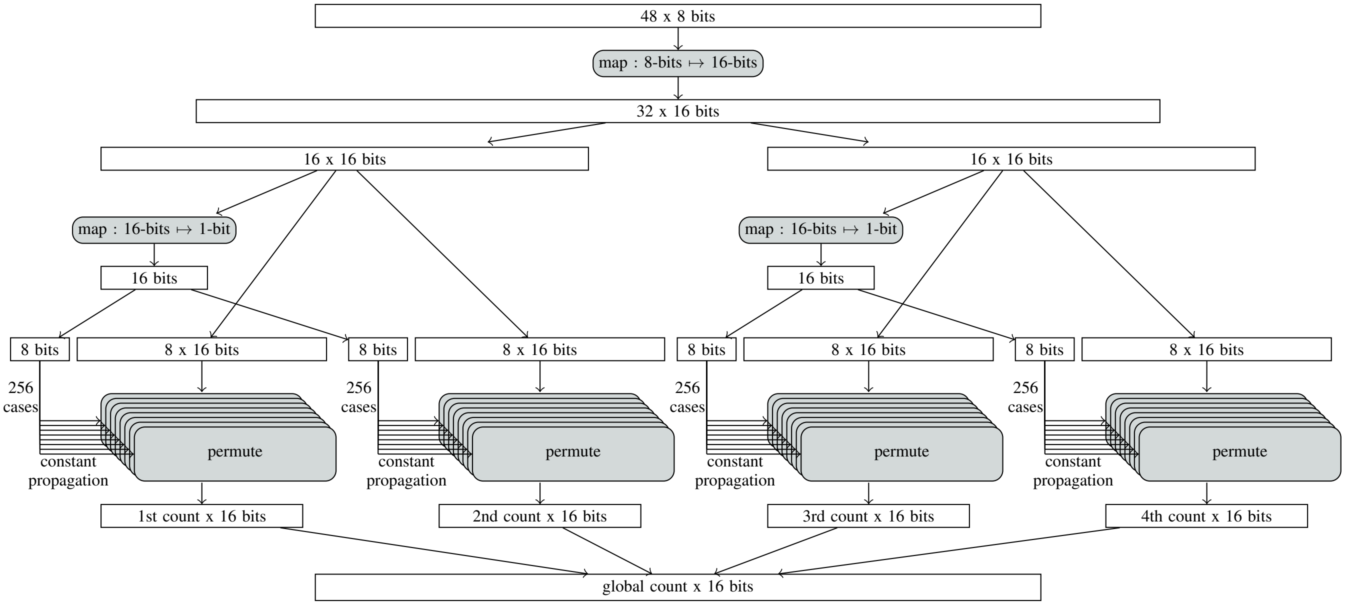

Figure 3. Hybrid representation of rejection sampling routine. White rectangles represent data. Grey curved rectangles represent circuits. Arrows represent high-level reasoning in the EasyCrypt Hoare logics.

specification is not discernable in the control-flow of the code, which in fact is straight-line;

2) There is no uniform vectorized computation pattern, as the implementation uses SIMD operations for various word sizes;
3) The code heavily uses permutation operations, and some of the used permutations are selected dynamically by computations that depend on the input data;
4) The code performs load/store operations at overlapping positions in input and output arrays; for the store operations, these positions are input-dependent.

As explained in the introduction, we overcome these challenges by carrying out the proof over a hybrid representation of the program, where we can recover the logical structure of the computation and, once that is done, apply automatic methods to resolving some of the low-level proof goals. Figure 3 shows schematically the hybrid reasoning that allows us to complete the proof.

We use Hoare logic to decompose the computation into smaller steps, that are carefully chosen to enable automatic proofs using circuit-based reasoning. We note that this high-level part to the proof is interactive: one needs to understand the program and break it down manually into pieces that can then be plugged into the circuit analysis backend. The manual interactive part then needs only to focus on the semantically interesting part of the proof—structuring the proof so as to reflect the top-to-bottom sequence of operations depicted in Figure 3—whereas the conversion to circuits and associated decision process that we describe next is fully automatic.

The greyed curved rectangles represent the parts of the program that are interpreted as circuits in the proof. As the program progresses, circuits are carrying out parallel or permutation computations across lanes of varying bit sizes: first 48x8 bits, then 48x16 bits, then 2x16x16 bits, then 4x(8+8x16) bits and finally 16-bit aligned results again. Some simple computations are handled directly in the EasyCrypt logics. However, the bulk of the proof is delegated to equivalence checking over circuits. The topmost circuit in Figure 3 corresponds to the computation performed in lines 8-11 of Figure 1. The two circuits in the midle correspond to lines 12-13. Finally, at the bottom, one can see 256x4 circuits that correspond to the possible permutations occurring in lines 24-32. Indeed, these permutation circuits become tractable only after a case analysis and constant propagation across four parallel lanes, which are broken down deductively. In Sections 5 and 7 we give more details about this proof. Before that, in what follows, we describe the framework that allows us to formalize it.

3. Background on EasyCrypt and Jasmin

EasyCrypt. EasyCrypt [4] is a proof assistant specialized to proving the correctness and security of of cryptographic constructions and their implementations. These objects can be specified in EasyCrypt using an imperative language called pWhile, which can be extended with arbitrary types and operators. The semantics of such extensions can be axiomatized, or they can be specified operationally using a functional language.

EasyCrypt supports interactive formal reasoning in various logics. In this work we are interested in the following two logics that permit reasoning about imperative programs:

- HL is a deterministic Hoare logic that allows to prove results on deterministic programs. Statements in this logic are of the form **hoare** [M.foo : P ⇛ Q], where M.foo is a procedure in module M, P is a precondition and Q is a post condition.

- RHL is a relational Hoare logic, that allows to prove relational results about the lock-step execution of a pair of programs. Statements in this logic are of the form equiv [M.foo ∼ N.foo : rP ⇛ rQ], where rP and rQ are now relational pre- and post- conditions that can refer to the initial and final states of both procedures.

In particular, for the classes of programs we consider in this paper, RHL can be used in conjunction with HL to prove results by transitivity: using a correctness result about a procedure P_1 in HL and an equivalence result between procedures P_1 and P_2 in RHL, one can prove a correctness result about P_2.

Additionally, we will rely on the ambient logic of EasyCrypt, which permits reasoning in high-order logic about properties of abstract and concrete types and operators, as well as manipulating results that describe properties of programs established using HL and RHL.

Proofs in EasyCrypt are written as a sequence of tactics, applying rules of the various logics. Additionally, some tactics permit applying an automatic decision procedure to decide a goal, e.g., by delegating it to an external SMT solver. The extension to EasyCrypt that we introduce in this paper enriches the set of tactics, introducing new ones that permit accessing a back-end for automated circuit-based reasoning.

Jasmin. Jasmin [6], [7] is a programming language and a set of accompanying tools designed to allow the efficient implementation and the formal verification of cryptographic primitives. It empowers programmers with precise control over low-level features of the assembly code produced by the Jasmin compiler, such as instruction selection and register spilling. Yet, the language provides high-level abstractions such as functions and arrays that help structuring programs and proofs. For instance, arrays have a functional semantics and functions are call-by-value, vastly simplifying modular reasoning. In accordance with the high-efficiency objective, most abstractions are eliminated at compile-time; the compiler checks that arrays are used linearly, which in turn allows to modify them in-place and avoid run-time copies.

Correctness of Jasmin Programs. The Jasmin compiler is certified by a Coq proof of correctness that guarantees the preservation of semantics from the Jasmin source to the target assembly code. This means that, whatever is proved about the functional behavior of the source Jasmin program, is propagated by the compiler to the generated assembly. A caveat to the above statement is that the correctness guarantee applies only if the source program is *safe*, a property that can be established using an automatic tool, also shipped with the Jasmin compiler (but not certified). Other properties of a source Jasmin program, such as the correct deployment of constant-time and speculative constant-time countermeasures, are also preserved by the compiler, and they can be proved at the source level using automatic type-based analysis tools also available with the compiler.

```
abstract theory BV.
 type bv.
 op size : int.  axiom gt0_size : 0 < size.
 op tolist : bv → bool list. op oflist : bool list → bv.
 op touint : bv → int. op tosint : bv → int. op ofint : int → bv.
 op get (b: bv) (n: int) : bool = List.nth false (tolist b) n.
 op msb (b: bv) : bool = List.nth false (tolist b) (size – 1).
 axiom size_tolist (bv : bv): List.size (tolist bv) = size.
 axiom tolistP (bv : bv) : oflist (tolist bv) = bv.
 axiom oflistP (xs : bool list) : size xs = size ⇒
                                  tolist (oflist xs) = xs.
 axiom touintP (bv : bv) : touint bv = bs2int (tolist bv).
 axiom tosintP (bv : bv) :
  (size = 1) ∨ let v = bs2int (tolist bv) in
               if (msb bv)
               then tosint bv = v – 2ˆsize
               else tosint bv = v.
 axiom ofintP (i : int) : ofint i = oflist (int2bs size i).
end BV.

abstract theory BVAdd.
 clone import BV.
 op bvadd : bv → bv → bv.
 axiom bvaddP (bv1 bv2 : bv) :
  touint (bvadd bv1 bv2) = (touint bv1 + touint bv2) % 2ˆBV.size.
end BVAdd.
```

Figure 4. Snippet of new QFBVA theory.

The functional correctness of Jasmin programs is proved by creating a model of the program in EasyCrypt: the Jasmin compiler can generate such a model automatically (this process is called extraction) as an imperative program that is syntactically very close to the original Jasmin code. The EasyCrypt model of the Jasmin program performs computations by using a set of operators that represent machine instructions; these operators are specified in an EasyCrypt library that is provided with the Jasmin compiler. The extraction to EasyCrypt and the correctness of the EasyCrypt library that captures the semantics of Jasmin operators are trusted: the assumption is that for any safe Jasmin program, whatever is proved in EasyCrypt about the extracted model applies to the Jasmin source. In what follows, when we refer to proving functional correctness properties of Jasmin programs, we refer to proofs carried out over the extracted EasyCrypt model.

4. Circuit-Based Reasoning in EasyCrypt

In this section we present the new features of EasyCrypt from the perspective of the user, and explain when and why user intervention is required to access circuit-based reasoning in a sound way. We will use examples related to the Jasmin programming language, but the same approach can be used for other programming languages.

4.1. Introducing Fixed-Sized Bit Vectors (BV)

As a unifying logical anchor for circuit-based proof goals we added to the EasyCrypt library a theory of bitvectors and bitvector arrays, as defined in SMT-LIB.[5] Figure 4

[5] https://smt-lib.org/

```
op xor_left (w1 : W8.t) = (w1 +^ (W8.of_int 42)) +^ (W8.of_int 213).

op xor_right (w1 : W8.t) = w1 +^ ((W8.of_int 42)) +^ (W8.of_int 213).

module M = {
  proc xor_left_proc (w1: W8.t) = {
    w1 ← w1 +^ (W8.of_int 42);
    w1 ← w1 +^ (W8.of_int 213);
    return w1;
  }

  proc xor_right_proc (w1: W8.t) = {
    var w2 : W8.t;
    w2 ← (W8.of_int 42);
    w2 ← w2 +^ (W8.of_int 213);
    w1 ←w1 +^ w2;
    return w1;
  }
}.
```

Figure 5. A toy example. Operator +^ denotes XOR over bytes (W8.t).

shows a snippet of the library code. This theory, called QFBVA, is not a contribution in itself, as it is a straightforward handwritten restatement of relevant parts of the SMT-LIB theories in EasyCrypt, with some extensions to allow interfacing the circuit-based reasoning backend. However, it does serve multiple useful purposes. Firstly, it creates a new set of EasyCrypt logics that can be directly translated to bit-vector operations natively supported by tools such as Z3 or Boolector. Secondly, it expresses the semantics of these bit-vector operations in the EasyCrypt core logics. This means that proof goals expressed using other EasyCrypt libraries, such as the Jasmin axiomatic semantics in EasyCrypt, can be restated in terms of SMT-LIB bit-vector theories. Both of these features were previously unavailable in EasyCrypt. Thirdly, and crucially for this work, it serves as an entry point that allows users to configure the tool with information about which data types and operations can appear in proof goals dischargeable using circuit-based analysis. Indeed, circuits are seen internally as EasyCrypt operators of type $\mathsf{bv}_m \mapsto \mathsf{bv}_n$ for arbitrary positive m and n. Here we take advantage also of the fact that bv_k types can be used interchangeably with Boolean lists of fixed size k, which follows from the bijection given by tolist and oflist. In what follows we will show how this connection can be made formal for arbitrary EasyCrypt types.

4.2. Basic circuit building blocks

Circuits that represent complex EasyCrypt expressions and whole (loop-free) programs are built modularly from an extendable set of basic circuits that we will call *gadgets*. The EasyCrypt circuit-based backend is bootstrapped with a core set of gadgets that covers the operations in QFBVA for bit vectors of arbitrary size, plus some useful modular buiding blocks such as sub-word and sub-array multiplexers, sub-word permutations, etc. Circuits are built by composing these basic building blocks, connecting output wires from one gate to input wires in the next gate in the obvious way. The composition machinery is generic with respect to word sizes, but relatively straightforward.

However, the QFBVA library is limited and does not capture all processor instructions, namely SIMD-style operations. For this reason, we provide an alternative means to add a new gadget to the EasyCrypt library and hardwire it (for the moment without any proof) as the circuit-based semantics of an EasyCrypt operator. To this end, we created a simple domain-specific language to externalize the specification of processor instructions. This is a functional language, where bit vectors of fixed size are native types, and where basic arithmetic operations and mechanisms for constructing and deconstructing bit vectors are polymorphic. There is also native support for map operations over sub-vectors, enabling very compact descriptions of vectorized instructions. We give two examples of the semantics of two x86-64 AVX2 instructions specified in this simple language. Variables can be annotated with their bit-vector sizes, notation w[@64|i] refers to the i-th 64-bit subword in w, whereas add smul are native arithmetic operations.

```
VPERMQ(w@256, i@8) → @256 =
 let permute (i@2) = w[@64|i] in
 concat⟨64⟩(permute(i[@2|0]),permute(i[@2|1]),
             permute(i[@2|2]),permute(i[@2|3]))

VPMADDWD_16u16(w1@256, w2@256) → @256 =
 map⟨32,8⟩(fun x@32 y@32 .
  add⟨32⟩(
    smul⟨16⟩(x[@16|0], y[@16|0]),
    smul⟨16⟩(x[@16|1], y[@16|1])), w1, w2)
```

EasyCrypt is able to parse and typecheck specifications of instructions in this language and, if they are accepted, to automatically generate circuits for them which are added to the list of available gadgets. We have used this mechanism to specify reusable gadgets for many x86-64 AVX2 instructions and ARM-m4 instructions that are required for our use cases. These are available in the EasyCrypt distribution. We heuristically validated these gadgets by fuzzing them with respect to baremetal instructions.

4.3. From models of Jasmin programs to circuits

Our goal is to enable the translation of a small but very useful class of Jasmin programs into a circuit representation. Intuitively, this class corresponds to loop-free programs whose input and output can be seen as Boolean lists of fixed length, and for which every computational step can be represented by one of the gadgets known to EasyCrypt.

To allow the user to access the circuit-based analysis backend, we have therefore introduced new EasyCrypt directives that allow the user to dynamically construct a set of *bindings* that tells the tool how arbitrary EasyCrypt types and operators can be mapped to bit vectors and circuit gadgets, respectively. We will explain these below, referring to the toy example in Figure 5.

Binding types. A binding for a type is always done via the QFBVA theory. We give an example for the type of Jasmin bytes W8.t.

```
bind bitstring W8.w2bits W8.bits2w
               W8.to_uint W8.to_sint W8.of_int W8.t 8.
```

This directive informs EasyCrypt that the user claims and is willing to prove that the W8.t type is a sound instantiation of bit vectors of size 8. To this end, the user provides instantiations for all the operators required to make the abstract BV theory from Figure 4 concrete, i.e., size = 8, bv = W8.t, tolist = W8.w2bits, oflist = W8.bits2w, touint = W8.to_uint, tosint = W8.to_sint and ofint = W8.of_int. The user is then asked to prove that these instantiations satisfy the properties (stated as **axiom** in the abstract theory, but discharged as lemmas upon instantiation) that are required to hold. We note that such properties imply that there is a bijection from values in the type to Boolean lists of length 8, and that the intepretation of values in the type as integers is also well defined. The latter is important for soundly binding QFBVA operators.

Using this mechanism we can introduce bindings for any bit-vector type and array thereof. In particular, we can easily map Jasmin arrays of words to bit vectors as follows.

```
bind array Array2.”_[_]” Array2.”_[_←_]”
           Array2.to_list Array2.of_list Array2.t 2.
```

In this case, the user needs to provide the usual get/set operators, plus a conversion to and from lists of the content type. Standard properties associated with array theories need to be justified by the user, such as extensional equality and get/set interaction. We note that such bindings are polymorphic in the content type, so they only need to be done once per array size.

Binding operators. We provide two directives that allow binding an EasyCrypt operator to a gadget, depending on whether the circuit-based semantics is established via the QFBVA gadgets, or via the domain-specific circuit specification language. For both, the binding will only work if the operator is of type $A \mapsto B$, where both types A and B are already bound to bit-vector types.

When the semantics of an operator is isomorphic to one of the already available QFBVA operators for the bound types, the user can proceed as follows.

```
bind op W8.t W8.(+^) ”xor”.
```

This EasyCrypt directive binds the XOR operator over W8.t, written +^, to the native xor circuit over bit vectors of size 8 via the QFBVA semantics. Indeed, to successfully complete the binding, the user is asked to prove that the semantics of +^ coincides with that specified in QFBVA. We note that this type of binding allows circuit-based reasoning over the W8.t type and, simultaneously, it allows a direct mapping of proof goals over W8.t to an external SMT tool that supports QFBVA. We also note that, conceptually, this binding does not introduce any additional assumptions wrt to the gadget library: the operation over W8 is simply reusing the assumption that the pre-existing gadget for XOR over bit-vector types is correct.

For more complex instructions that do not have a direct mapping to QFBVA operations, we allow the user to define a binding as follows.

```
bind circuit VPMADDWD_256 ”VPMADDWD_16u16”.
```

This binding informs EasyCrypt that it has the user's blessing to replace every occurrence of operator VPMADDWD_256 with the corresponding gadget generated from the specification of VPMADDWD_16u16. In this case, the binding directive is indeed introducing a new assumption in the proof: that the semantics of the EasyCrypt operator that represents the VPMADDWD_256 instruction matches the semantics of the circuit named VPMADDWD_16u16.

To further clarify the difference between **bind op** and **bind circuit**, consider the xor_left operator in Figure 5. One can obtain a circuit for it via the binding of the +^ operator over W8.t to the corresponding QFBVA circuit, and then relying on the EasyCrypt machinery to construct the circuit for the composed expression based on the pre-existing bit-vector XOR gadget. This would not introduce new assumptions.

Alternatively, one could specify a circuit using our domain-specific language as

```
XOR_LEFT8(w@8) → @8 = xor⟨8⟩(xor⟨8⟩(w, 42@8), 213@8)
```

Binding this new gadget directly to xor_left using **bind circuit** would correspond to introducing a new axiom in the proof. In our current development we use **bind circuit** to enrich the set of gadgets with support for SIMD operations in an agile way. In future work we plan to automatically synthethise EasyCrypt operators from the specification language description, and then ask the user to prove that the semantics match. I.e., rather than introducing a new assumption, we will ask the user to prove it as a lemma.

We defer to Section 6 a more detailed explanation of how a full circuit for a complex EasyCrypt expression or program is generated. We conclude this section by describing the set of EasyCrypt proof goals that a user can delegate to circuit-based analysis.

4.4. Automated proofs of EasyCrypt judgements

The binding mechanism we described above gives rise to a hybrid representation of a program, and enables a fully automated method for proving a class of EasyCrypt proof goals that can be restated as the extensional equality of two circuits over fixed-sized bit vectors, possibly over a restricted set of inputs. In other words, this automatic method works by restating EasyCrypt judgements as equivalences between circuits.

Operator equivalence. The simplest EasyCrypt judgement one can discharge in this way is an ambient logic formula stating equivalence between two operators $f_1, f_2 : A \mapsto B$ of the form:

$$\forall x,\ P(x) \ \Rightarrow\ f_1(x) = f_2(x) \qquad (1)$$

where P is a Boolean function that constrains the set of inputs over which the equivalence must hold.

The soundness of using circuit-based equivalence checking to discharge this class of goals follows trivially from the fact that the circuit reasoning backend of EasyCrypt and the binding framework we described above are designed (and trusted) to construct, for $i \in \{1, 2\}$, circuits C_i such that

$$\forall x, (\mathsf{tolist} \circ C_i \circ \mathsf{oflist})(x) = f_i(x)$$

where tolist and oflist are the bijection proved on binding.

For example, one can prove the following equivalence between operators xor_left and xor_right in Figure 5

```
∀ (w: W8.t) : xor_left w = xor_right w
```

by using the tactic **bdep solve**, which recognizes the goal, converts both sides of the equality to a circuit, and proves equivalence.

Program correctness. The extension to programs is straightforward. Consider a loop-free deterministic EasyCrypt program S that uses only types and operators that have been bound to the circuit backend. Consider also a proof goal of the form

$$\forall x, \{arg = x \wedge P(x)\}\, S\, \{res = f_1(x)\} \tag{2}$$

i.e., a Hoare triple where the pre-condition fixes the input value to some arbitrary constant x that satisfies a predicate P, and where the post-condition requires that the result is equal to $f(x)$. Then, because S is loop-free, it is clear that the previous proof goal can be re-expressed in EasyCrypt as follows:

$$\forall x, \{arg = x \wedge P(x)\}\, \textbf{skip}\, \{f_2(x) = f_1(x)\}$$

where the function f_2 is constructed automatically by the weakest precondition calculus in EasyCrypt. It therefore follows that, for such programs, goals of the form given in Equation 2 can always be transformed into goals of the form given in Equation 1.

For example, suppose that we want to prove that the procedure xor_left_proc in Figure 5 satisfies the following Hoare triple, where w is a logical variable quantified universally:

```
hoare [ M.xor_left_proc : w = w1 ⇒  res = xor_left w].
```

Then one can use the following tactic

```
bdep 8 8 [w] [w1] [w1] xor_left_spec predT_W8.
```

where EasyCrypt is asked to prove that, when seen as a circuit, the procedure code is equivalent to the circuit corresponding to the operator xor_left_spec. Here, both circuits are defined as having inputs (w1) and outputs (again w1) bit-vectors of size 8, and equivalence must hold for all inputs (predT_W8 is the predicate that accepts all bit-strings of size 8). Then, EasyCrypt automatically generates the circuits and, after checking circuit equivalence, asks the user to prove the following two implications:

```
w = w1{hr} ⇒  predT_W8 (W8.bits2w (w2bits w)) ∧ w = w1{hr}

∀  (w1_0 : W8.t),
 xor_left (W8.bits2w (w2bits w)) = W8.bits2w (w2bits w1_0)
   ⇒  w1_0 = xor_left w
```

The first implication requires the user to show that the Hoare triple pre-condition implies the circuit-equivalence precondition (which in this case is trivial). The second implication requires the user to show that the equivalence shown at the circuit level implies the post-condition of the Hoare triple. Again, in this case the implication is trivial but, in general, the circuit-based equivalence is referring to a circuit that operates over the combined bit-string of all program inputs (in this case there is only one) and produces a bit string that is interpreted as the concatenation of all program outputs (again here there is only one).

Intuitively, every application of the **bdep** tactic over (relational) Hoare triples, can be seen as an application of the consequence rule, where the user is asked to prove that a Hoare triple automatically proved using the circuit-based backend implies the Hoare triple that the user wants to discharge. I.e., when proving $\{P\}\, C\, \{Q\}$ using **bdep**, if the tactic is successful it returns $\{P'\}\, C\, \{Q'\}$ as a proved result, where P' and C' refer to the bit string representation of the program state and depend on the analysis that was carried out at the circuit level. It remains to prove in EasyCrypt that $P \Rightarrow P'$ and $Q' \Rightarrow Q$. These two implications are usually straightforward to discharge, and much simpler to prove than the original proof goal.

Program equivalence. Finally, by the same reasoning, we can justify discharging relational Hoare triples of the form:

$$\forall x, \left\{ \begin{array}{c} arg_1 = arg_2 \wedge \\ arg_1 = x \wedge P(x) \end{array} \right\} S_1 \sim S_2 \left\{ \begin{array}{c} \\ res_1 = res_2 \end{array} \right\} \tag{3}$$

Indeed, the EasyCrypt weakest precondition calculus can infer f_1 and f_2 corresponding to S_1 and S_2 and reduce both programs to **skip**, so that the relational Hoare triple can be restated as an implication.

For example, suppose that we want to prove that the procedure xor_left_proc is equivalent to the procedure xor_right_proc in Figure 5, i.e., that the following relational Hoare triple holds:

```
equiv [ M.xor_left_proc ∼  M.xor_right_proc :
    w = arg{1} ∧ arg{1} = arg{2} ⇒  res{1} = res{2} ].
```

Then one can use the following tactic

```
bdepeq 8 [w1] [w1] {8 : [w1 ∼  w1]} predT_W8.
```

where EasyCrypt is asked to prove that, when seen as circuits, the code of both procedures is equivalent. Again, both circuits are defined as having inputs (w1) and outputs (again w1) bit-vectors of size 8, and equivalence must hold for all inputs (predT_W8 is the predicate that accepts all bit-strings of size 8). Then, EasyCrypt automatically generates the circuits and, after checking circuit equivalence, asks the user to prove two natural implications: 1) that the precondition implies that inputs on both sides are initially the same and predT_W8 holds initially; and 2) that equality of the circuit outputs implies the relational postcondition.

Proving circuit equivalence. In theory, the framework above works for any decision procedure for circuit equivalence. In some of our examples, this follows directly from structural equivalence, i.e., it can be checked simply by comparing the representation of the circuit, which can be done with essentially no cost (cf. Section 6). When the circuits have a different structure, we carry out equivalence checking at the logical level by calling an external SAT solver (in the future we aim to experiment with various equivalence checking backends). Finally, in some cases that we will explain in the next section, the trivial approach of proving equivalence by exhaustive evaluation works best. Indeed, circuit equivalence checking does not always work satisfactorily when applied naively to the examples we address in this work. This happens when the the specification and/or the computation are not easily expressible as a reasonably-sized circuit. In the next section we describe our approaches to dealing with this problem.

5. Scaling by Using Hybrid Reasoning

In this section, we introduce a hybrid approach that addresses these scalability issues. Our approach, which integrates deductive verification, also overcomes the limitations of a purely deductive approach to functional correctness, which also doesn't scale well, as was reported in prior work [11].

Dependency analysis for exhaustive evaluation. While blunt, the exhaustive evaluation approach can be refined so that it can be used in a sound and scalable way to address a common pattern that occurs in post-quantum cryptography: serialization functions that compress algebraic structure representations into byte arrays.

At a high-level, these serialization functions are computing a map: the algebraic representation can be seen as a list of fixed size (order 1K is typical) of finite-field elements. Each field element is compressed/decompressed using the same algorithm to/from a small number of bits (e.g., 1, 4, 10 and 12 bits are values that occur in ML-KEM 768). Proving such functions deductively requires extensive boiler-plate code because of the bit-level semantics, particularly in the case of SIMD implementations that totally obfuscate the logical structure of the algorithm. As an example, consider the EasyCrypt specification of compressing an integer representing an element in $\mathbb{F}_q$ by rounding it to another integer that fits into d bits.

```
op compress(c : int) : int = (c % q * 2^d + (q / 2)) / q % 2^d.
```

This is a very high-level specification that can be evaluated as an EasyCrypt operator—EasyCrypt supports evaluation of int expressions—but for which there is no obvious circuit representation, as it is expressed over native EasyCrypt integers. Indeed, in previous proofs of ML-KEM in Easycrypt [11], [12], showing that compression is correctly implemented by a sequence of processor instructions required architecture-specific reasoning, as different sequences of processor instructions are used for different compression sizes. This pattern occurs also in other variants of ML-KEM not covered by the proofs in [11], [12], and it occurs as well in implementations of ML-DSA.

The new circuit-based reasoning backend in EasyCrypt permits proving such functions with a fraction of the effort, relying almost entirely on automation. Our approach is as follows. We introduce a circuit-based dependency analysis mechanism that, given a circuit, infers for each output bit the input bits on which it depends. Based on this, the user is offered a variant of the **bdep** tactic that takes as inputs: 1) the EasyCrypt program; 2) the number of parallel *lanes* in which the computation is logically structured; 3) the specification of the computation done by one lane; and 4) the range of inputs over which the equivalence must hold. The EasyCrypt tactic then tries to discharge the goal as follows:

1) It converts the program into a circuit;
2) It runs the dependency analysis to confirm that the full circuit decomposes into a parallel computation as claimed by the user;
3) It extracts the circuits for each of the parallel lanes;
4) It runs pairwise equivalence checks between lanes to ensure that they all compute the same function; and
5) It checks that the first lane is equivalent to the spec by exhaustive evaluation for all possible inputs of such lane.

We note that step 4 above is usually trivial because the circuits for all the lanes are identical: in fact, in this case the circuit representation we use permits recognising this at essentially zero cost [16]. We also note that step 5) can be replaced by any equivalence checking procedure and, in particular, if the lane function specification can be converted to a circuit, then it can also be discharged by circuit equivalence. Indeed, EasyCrypt offers a variant of the **bdep** tactic that works in this way, so the user can choose which decision procedure to use.

Clearly, this simple strategy can be used to discharge extensional equivalence goals of the form given by Equation 1, when f_1 and f_2 are map computations. More precisely, the **bdep** tactic as described above can be used to discharge the functional correctness of the compression algorithm via a judgement of the form:

$$\begin{aligned}\forall x,\ \forall i \in [0:n], P([\mathsf{chunk}(16,x)]_i) \Rightarrow \\ \mathsf{chunk}(10, f_2(x)) = \mathsf{map}\ g\ (\mathsf{chunk}(16,x))\end{aligned}$$

where g denotes the lane function, n denotes the number of lanes, P denotes the precondition that must hold on every lane input, and $\mathsf{chunk}(k,x)$ breaks down x into sub-bit-vectors of size k (we require that k divides the size of x).

To further illustrate this kind of reasoning, let us revisit the example in Figure 5 and the following Hoare triple:

```
hoare [ M.xor_left_proc : w = w1 ⇒ res = xor_left w].
```

An alternative way to discharge this goal is to observe that $213u8 \oplus 42u8 = 255u8$, and use the **bdep** tactic as follows to leverage the fact that XOR is computed in parallel lanes of width 1:

```
op xor1_bool (b: bool) = b ^^ true.

bdep 1 1 [w] [w1] [w1] xor1_bool predT_bool.
```

This will cause EasyCrypt to check that, indeed, the program circuit decomposes into 8 lanes of width 1, and that each of them computes the circuit xor1_bool. As side conditions the user is again asked to prove that the trivial pre-condition holds (in this case for all of the 8 lanes), and that the established circuit-based post-condition implies that the program is indeed computing xor_left. This latter implication takes the following form:

```
∀ (w1_0 : W8.t),
  map xor1_bool (map bits2bool (chunk 1 (w2bits w))) =
    map bits2bool (chunk 1 (w2bits w1_0)) ⇒ w1_0 = xor_left w
```

Observe that, albeit straightforward to prove in EasyCrypt, this implication is no longer trivial and, indeed, there is a tradeoff: while lane-wise equivalence permits scaling the circuit-based analysis, using it may require massaging the proof goal in order to establish that, indeed, the computation one is performing is structured as a map.

We will showcase and evaluate the benefits of the various forms of the **bdep** tactic in Section 7. For now, we will look at yet another variant of the tactic that also leverages dependency analysis, but for relational goals.

Dependency analysis in relational goals. Relational Hoare logic is a powerful tool. Judgments in this logic can be used to propagate a functional correctness result from a program S_1 to a program S_2 by transitivity. I.e., if we know that S_1 satisfies Hoare triple $\{P\}\, S_1\, \{Q\}$ and that S_1 and S_2 satisfy the relational contract $\{arg_1 = arg_2\}\, S_1 \sim S_2\, \{res_1 = res_2\}$, then the EasyCrypt logic allows us to derive that $\{P\}\, S_2\, \{Q\}$ holds. We note that if the effort required to carry out the relational step is small, then this permits amortizing the effort involved in proving the first implementation functionally correct. Unfortunately, although the relational Hoare logics of EasyCrypt permits proving these transitivity goals interactively, they were designed for game-hopping security proofs where the control-flow structure and operations used by both programs are very close.

In functional correctness proofs, however, it is often the case that reference and optimized programs are not easy to analyse in lockstep. For example, when using SIMD operations, or when optimizing programs for different memory usages, the control flow and instruction sets used by the programs can be quite different. In prior proof of MLKEM [11], [12], for example, the strategy of using transitivity and relational Hoare logic to derive a proof for an AVX2 implementation from an existing proof for a reference implementation was not effective. Indeed, for many cases, including the serialization functions discussed above, it was easier to prove independently that the AVX2 functions satisfied a high-level functional specification.

However, as seen above, when using a circuit representation, it is often possible to recognize a common logical structure in both programs. Indeed, if S_1 and S_2 are performing parallel computations across n lanes with different instruction sets, and using different control-flow or scheduling, the parallel structure can be recognized with dependency analysis on each side. Moreover, the circuits that are being computed by each lane can be easily extracted. Then, equivalence can be proved lane by lane.

Another variant of the **bdep** tactic in EasyCrypt (already briefly showcased above) permits discharging relational goals using precisely this strategy. For example, referring back to Figure 5, one could revisit the relational goal:

```
equiv [ M.xor_left_proc ∼ M.xor_right_proc :
    w = arg{1} ∧ arg{1} = arg{2} ⇒ res{1} = res{2} ].
```

Then, one can use the **bdepeq** tactic differently as follows:

```
bdepeq 1 [w1] [w1] {1 : [w1 ∼ w1]} predT_bool.
```

From the user perspective, this tactic produces exactly the same outcome as if one asked equivalence to be checked over the whole circuit. Indeed, the user is not even asked to specify what the lane functions are doing, or even if they are all doing the same thing. The only requirement is that both circuits decompose into lanes of size 1, and that such per-lane equivalence holds.

Other variants of the **bdep** tactic can be added in a straightforward way, for example to establish the relational goal simultaneously with the functional correctness goal on one of the sides.[6] The high-level EasyCrypt logics are expressive enough to capture and reason compositionally about judgements of the form:

$$\forall x, \left\{ \begin{array}{c} arg_1 = arg_2 \\ \wedge \\ arg_1 = x \end{array} \right\} S_1 \sim S_2 \left\{ \begin{array}{c} res_1 = res_2 \\ \wedge \\ res_1 = f(x) \end{array} \right\}$$

We have used the approach we just described to construct a new (shorter) proof for the Jasmin reference implementation [11], [12] of the compression function in ML-KEM, while at the same time proving it equivalent to a faster AVX2 version developed specifically for this work. Moreover, we used the transitivity approach to derive functional correctness results for ARM-m4 (a 32-bit architecture) code from x64-64 implementations. We say more on these results in Section 7.

We conclude this section with an example of how deductive reasoning and high-level EasyCrypt tactics can be used in combination with circuit-based reasoning to tackle challenging proof goals that were previously out of reach.

Scaling by modular deductive reasoning. Let us return to the motivating example we presented in Section 2. We hope the reader will agree that, looking at Figure 1, it is not straightforward to see how the circuit-based reasoning tactics we presented above can be used prove functional correctness: the program does not seem to carry out parallel computation and, indeed, the program and the specification both give rise to large and complex-structured circuits.

[6] Future work in EasyCrypt will certainly broaden the range of tactics relying on the circuit-based machinery and dependency analysis in particular. For example, dealing with container initialization patterns, where the same function is being computed on each lane, but with a lane-dependent behavior can be handled by combining dependency analysis and circuit decomposition with constant propagation of the lane index.

However, our approach consists of using the EasyCrypt high-level logics to break the problem into smaller parts that can, in fact, be proved using circuit-based analysis. We showed this in diagram form in Figure 3. The first step is to use the sequence rule of Hoare logic to break the proof goal into five parts: 1) reading 48x8 bits from stack into two 256-bit registers; 2) rearranging the input into 32 candidates, arranged as 2 sets of sixteen 16-bit words 3) computing the 2x16-bit mask that encodes the accepted and rejected candidates 4) move the accepted candidates into consecutive lefmost positions of SIMD registers, using the above bit masks and memory tables that define the necessary permutations 5) computing the number of accepted candidates and offsets for storage, and completing the computation by writing the results to the output buffer.

After this split is done, the Hoare triples that correspond to steps 1) and 5) are discharged interactively, since their logic is naturally expressed in the semantics of Jasmin formalized in EasyCrypt. For example, proving correct computation of the number of accepted candidates in goal 5) is straightforward because the semantics of the #POPCNT_64 instruction is exactly defined as counting the number of 1 bits in the input, when it is seen as a list of bits. Then, the remaining steps can all be seen as different instances of the circuit-based reasoning illustrated above, as follows:

- step 2) can be seen as a parallel computation that is repeated four times, as described in Figure 2. This corresponds to the top level of circuitry in Figure 3. Moreover, the specification for this computation is naturally expressed as *chunking* the input into 12-bit words and *chunking* the output into 16-bit words, which we have seen above to be the natural language of circuit-based reasoning. Furthermore, the specification of the map computation being done for each of the output chunks is simply the padding with 4 zero bits of the corresponding input chunk;
- in step 3), which corresponds to the intermediate circuits in Figure 3, the computation of the masks can again be easily seen as a parallel computation of the Boolean predicate that checks whether a candidate is in the acceptable range. So, in this case, the parallel computation is mapping 12-bits to 1-bit on each lane;
- finally, step 4) is the most complex of all because the program is actually using input-dependent indices to access memory tables that define which permutation should be computed to put the accepted candidates in their correct place. Working with a circuit representation of these table accesses would not be feasible. However, we can leverage in the proof the same intuition that makes the SIMD implementation work: the table of possible permutations has only 256 entries, a small number. This means that we can use the high-level reasoning of easycrypt to restate step 4) as 256x4 Hoare triples, each corresponding to one possible index to the permutation table. Furthermore, for each of these goals, the table access can be removed by constant propagation and the computation expressed as a small circuit that computes the permutation. From here, circuit-based reasoning takes over and permits discharging the proof goals automatically. This circuit-based reasoning is illustrated at the bottom of Figure 3. We call this proof strategy *exhaustive partial evaluation*.

To the best of our knowledge, the above proof strategy is novel in that it crucially relies on hybrid reasoning in different computation models and logical frameworks to remove a major hurdle in the proof of functional correctness of optimized ML-KEM. Furthermore, it can also be used in other settings where rejection sampling is used, namely ML-DSA. We note, however, that although the above description may now seem intuitive, finding the best approach to formalizing the details was non-trivial, and it required several enhancements to the EasyCrypt front-end to allow expressing it in a reasonably compact way. We defer more details to Section 7 and describe the tool development work carried out in EasyCrypt next.

6. Implementation of New EasyCrypt Features

To support circuit-based reasoning in EasyCrypt, we developed a new OCaml module which introduces functionalities for constructing and manipulating circuits. This extension is designed to integrate incrementally with the existing EasyCrypt codebase, with a focus on modularity and reuse of existing infrastructure.

Circuit manipulation is implemented in a dedicated library based on And-Inverted Graphs (AIG), a widely used representation in equivalence checking (e.g., in tools like ABC [17]). AIG circuits provide a simple yet expressive foundation for dependency analysis and functional equivalence checking [18].

On top of this AIG-based representation, we construct standard ALU circuits for arithmetic and logical operations, following the semantics defined by the SMT-LIB QFABV theory [19]. These ALU circuits can be translated into AIG circuits (for dependency analysis and equivalence checking) or directly into SMT-LIB formulas.

The circuits module is integrated into EasyCrypt's reasoning framework by adding the ability to translate EasyCrypt formulas (representing loop-free computations) into circuits. This translation process traverses expressions recursively, replacing operations with corresponding gadgets (as mentioned in Section 4) and composing subcircuits accordingly. The composition process abstracts over the raw AIG structure while preserving type information to connect the AIG representation with the high-level reasoning used in EasyCrypt tactics.

Using this infrastructure, we implemented the new family of **bdep** tactics for circuit-based dependency analysis and equivalence checking. These tactics analyze dependencies by traversing a circuit's outputs and determining the input bits that affect each output. This allows checking whether a circuit decomposes into independent lanes, as claimed by the tactic user. Lane equivalence within the same program is typically checked via structural equality of their functional representations. For more complex equivalence proofs, we employ a SMT solver, formulating the problem as checking whether a solution exists for the predicate

$$P(x) \wedge (C_1(x) \neq C_2(x))$$

where C_1 and C_2 are the circuits to be compared under a precondition P. An unsatisfiability (UNSAT) result confirms equivalence.

To facilitate reasoning about programs before their circuit translation, we extended EasyCrypt with new high-level program transformation tactics, including **proc change** and **proc rewrite**. These allow replacing program fragments with equivalent implementations, ensuring that all operations map to defined circuit gadgets. Improvements in loop unrolling and constant propagation further streamline program transformations, effectively eliminating integer variables in circuits.

Trusted Code-Base (TCB). Our design ensures a minimal and well-defined impact on the existing EasyCrypt Trusted Code Base (TCB). The original TCB includes: the core higher-order logic kernel, the core program logic kernel, the translation to SMT solvers, and the solvers themselves (via the Why3 framework). All proof tactics in EasyCrypt produce proof witnesses, and the checking of these witnesses reduces to the components listed above.

To support circuit reasoning, we extend the TCB with the OCaml library that implements AIG-based circuit representations, dependency analysis, and functional equivalence checking. The semantics and correctness of the ALU circuits we define——following the QFABV SMT-LIB theory——are also explicitly included in the TCB. Furthermore, their translation to SMT-LIB is trusted due to its straightforward, one-to-one mapping with SMT operators.

This extension does not alter the core logic or proof system of EasyCrypt. The use of SMT solvers for equivalence checking aligns with longstanding EasyCrypt practice and does not expand the TCB beyond its historical scope.

As described in Section 4, a key design choice is the ability for users to bind EasyCrypt bitstring operators to circuits. We conclude this section with a short recap of the impact of these user-driven bindings on the TCB.

Bindings established using **bind op** via the QFABV circuit semantics do not extend the TCB because they require correctness proofs, making them verifiable extensions. As such, users can define increasingly complex circuits that are still subject to EasyCrypt's proof discipline and automated checking infrastructure. Bindings established using **bind circuit** and the domain-specific circuit description language do extend the TCB: the user is adding the assumption that the semantics of the EasyCrypt operator and provided circuit match. While this currently overlaps with EasyCrypt's existing constructs, our long-term goal is to unify this mechanism with EasyCrypt and Jasmin's ISA semantics. Such consolidation will further reduce the TCB by removing redundant semantic representations.

Finally, because our extension is modular and optional, it does not impact EasyCrypt developments that do not involve circuits. All new functionalities act as an API layer that automates proof obligations, but they do not introduce any unverified components into the trusted core.

7. Use Cases and Experimental Evaluation

We now present the results we obtained with the new proof automation features introduced in EasyCrypt. We focus on ML-KEM because this allows us to establish a direct comparison to a previously existing formally verified open-source implementation [11], [12] which will serve as baseline. All examples in this section are included in the artifact that accompanies the submission.

7.1. Fast AVX2 rejection sampling in ML-KEM

In Sections 2 and 5 we already gave an overview of our proof strategy for our motivating example: the filtering routine used in the AVX2 implementations of ML-KEM. Figure 6 shows highlights of the proof script, which relies on hybrid reasoning in and where the steps in the proof map to those described in Section 5.

The **seq** ^⟨pattern⟩ tactic allows breaking the program at the first occurrence of ⟨pattern⟩ using the sequence rule for Hoare logic and decompose the proof goal into two Hoare triples by providing a bridging post/pre-condition as argument. One can clearly see for step 2) the use of **bdep** to recover the map from 12-bit chunks to 16-bit chunks, which is a trivial extension by adding zero bits, but carried out by a complex sequence of SIMD instructions in the code. For step 3) the same tactic is used to prove that every individual bit in good holds the result of comparing a candidate to the threshold. In both cases the computations are seen as a maps and proved correct using dependency analysis and circuit equivalence. Finally, for step 4), we show how in the middle of the proof, one can define a specification P for the computation that will be partially evaluated inside the **bdep** tactic for each value of input g, thereby yielding a fixed permutation that can be used as target for equivalence with the code precisely on the same value of g. We recall that in this case the code is performing a table access based on this value to fix the argument of a SIMD permutation operation, which is again resolved by constant propagation before converting the program to a circuit. Overall, the proof comprises 350 lines of code, where the most verbose parts correspond to the deductive reasoning that justifies writing to the output array in overlapped regions.

7.2. Faster compression in ML-KEM

We have introduced the notion of compression of a finite field element in Section 5. We return to that example to illustrate what circuit-based reasoning buys for proof repair when formally verified code is modified. The functional correctness proofs of ML-KEM for Jasmin code reported in [11], [12] were written in a purely deductive way and required deep algebraic reasoning about the following three steps: 1) a (floor) Barrett reduction step that takes a representation of a finite field element in the range $[-2^{15}, 2^{15}-1]$, i.e.

```
(* Step 2: extracting all the 12–bit words from the input buffer *)
seq^g0←{2} & –1 : (∀ i, 0 ≤ i < 32 ⇒
        extract_512_16 (concat_2u256 f0 f1) (16 * i)
            = zextend_12_16 (sliceget_8_12_56 buf (12 * i))).
– bdep 12 16 [_buf] [buf] [f0; f1] zextend_12_16 predT_W12.

(* Step 3: parallel comparison, computing good *)
seq^good← : (#pre ∧
 ∀ i, 0 ≤ i < 32 ⇒ good[perm i] ⇔
     extract_512_16 (concat_2u256 f0 f1) (16 * i) <s 3329).
– bdep 16 1 [_f0; _f1] [f0; f1] [good] ltq predT_W16 perm.

(* Step 4: Exhaustive partial evaluation and permutation *)
pose P (o : int) (g : W8.t) (f : W256.t) (f_0 : W128.t) ←
  let w = pmap (fun i ⇒ if g[i]
              then Some (extract_256_16 f (o + 16 * i))
              else None) (iota_ 0 8) in
  all (fun i ⇒ w[i] = extract_128_16 f_0 (16 * i)) (iota_ 0 (size w)).

seq^f0_0← : (#pre ∧ P 0 good0_0 f0 f0_0).
– bdep bitstring [f0] [P 0 good0_0 f0 f0_0] good0_0.
seq^f0_1← : (#pre ∧ P 128 good0_1 f0 f0_1).
– bdep bitstring [f0] [P 128 good0_1 f0 f0_1] good0_1.
seq^f1_0← : (#pre ∧ P 0 good1_0 f1 f1_0).
– bdep bitstring [f1] [P 0 good1_0 f1 f1_0] good1_0.
sp 1; seq^f1_1← : (#pre ∧ P 128 good1_1 f1 f1_1).
– bdep bitstring [f1] [P 128 good1_1 f1 f1_1] good1_1.
```

Figure 6. Highlights of the proof script for the filtering routine.

the full signed range of 16-bit words, to the range $[0, q]$; 2) a conditional subtraction step that further reduces the range to $[0, q)$; and a compressing step that computes rounding to $d = 10$-bits as specified by ML-KEM.

In this paper we consider an alternative, more efficient, version of this serialization process, which we now explain.[7]

More on compression. The standard techinique to compute $\lfloor a2^d/q \rceil$ is to apply Barrett reduction to $a2^d$: one chooses a precision $\mathtt{R} = 2^k$ and computes $\lfloor a \lfloor \mathtt{R}/q \rceil / (\mathtt{R}/2^d) \rceil$. As long as $|a|$ is smaller than a certain value $\mathtt{B}$ depending on $d, \mathtt{R}$, and q, we have $\lfloor a2^d/q \rceil = \lfloor a \lfloor \mathtt{R}/q \rceil / (\mathtt{R}/2^d) \rceil$, where the latter expression is straightforward to implement using processor instructions. However, small differences in the value $\mathtt{B}$ can make a big difference when it comes to implementing the algorithm in a given instruction set.

We illustrate this by considering a signed variant of the compression procedure. We first observe that $\lfloor (a + lq)\, 2^d/q \rceil \bmod 2^d = \lfloor a2^d/q \rceil \bmod 2^d$ for an arbitrary integer l. This implies we can work with the signed integers in the range $\left[-\frac{q-1}{2}, \frac{q-1}{2}\right]$ instead of the unsigned ones in the range $[0, q)$, halving the requirement of $\mathtt{B}$. The first modification of our compression procedure is therefore to replace the floor version of Barrett reduction with the rounding one, mapping elements to $\left[-\frac{q-1}{2}, \frac{q-1}{2}\right]$.

In theory, invoking [20, Proposition 1], $|a| < \frac{\mathtt{R}}{2^d(q-1)}$ suffices for concluding $\lfloor a2^d/q \rceil = \lfloor a \lfloor \mathtt{R}/q \rceil / (\mathtt{R}/2^d) \rceil$. Unfortunately, this analytical approach does not permit to easily justify the efficient implementation we seek in AVX2. Indeed, for $d = 10$, as $\log_2\left(|a| \cdot 2^d(q-1)\right) > 32$ for some $a \in \left[-\frac{q-1}{2}, \frac{q-1}{2}\right]$, we require $\mathtt{R} \geq 2^{33}$.

```
inline fn compress10_16x16_inline(
    reg u256 a b0 b1 b2 mask) → reg u256 {
    reg u256 p0 p1;
    p0 = #VPMULH_16u16(a, b0);
    p1 = #VPMULL_16u16(a, b1);
    p0 = #VPADD_16u16(p0, p1);
    p0 = #VPMULHRS_16u16(p0, b2);
    p0 = #VPAND_256(p0, mask);
    return p0;
}
```

Figure 7. Improved $\mathtt{Compress}_{10}$. Inlined 768/16 times round 768 values in $[-(q-1)/2, (q-1)/2]$.

A common way that practitioners use for sharpening the precision $\mathtt{R}$ is brute-force testing: one simply tests if for all a from a given range, the identity $\lfloor a2^d/q \rceil = \lfloor a \lfloor \mathtt{R}/q \rceil / (\mathtt{R}/2^d) \rceil$ holds. For the case $d = 10$, experiments show that $\mathtt{R} = 2^{32}$ is the minimum power of two satisfying $\lfloor a2^{10}/q \rceil = \lfloor a \lfloor 2^{32}/q \rceil / 2^{22} \rceil$ for all $a \in \left[-\frac{q-1}{2}, \frac{q-1}{2}\right]$. The algorithm that is justified in this way is given in Figure 7. We stress that, since the algorithm is finetuned by practitioners using this approach, there is no immediately available proof that can be formalized. Furthermore, formalizing such a proof would probably incur in an effort comparable to the proof constructed for the previous version of the algorithm, which used a different tuning.

Amortizing proof effort. Interestingly, using our circuit-based approach we are able to amortize the effort of proving the *reference implementation* of this algorithm and obtain a proof for the new one with minimal effort. Using the proof of the AVX2 implementation in [11] as a baseline (noting that it could be rewritten using circuit-based reasoning as well) one goes from a proof taking 650 lines of code to one taking 100 lines of code. The proof goes as follows. We first observe that, when proving the reference implementation of the compression routines, prior work established that the following EasyCrypt specification is equivalent to the mathematical specification of the rounding procedure.

```
op reduce(c : W16.t) : W16.t =
  let t = (sigextu32 c) * (W32.of_int 20159) in
  let t = (sra_32 t (W32.of_int 26)) in
  let t = (t * (W32.of_int 3329)) in
  let r = c – (W2u16.truncateu16 t) in r.

op csubq(c : W16.t) : W16.t =
  if (c < (W16.of_int 3329)) then c else (c – (W16.of_int 3329)).

op compress10(x : W16.t) : W10.t =
 truncate10
   (((((zeroextu64 x) `<<` (W64.of_int 10)) + W64.of_int 1665) *
       (W64.of_int 1290167)) `>>` (W64.of_int 32)).

op lane_func_compress10 = compress10 \o csubq \o reduce.
```

Here, sigextu32 sign-extends a 16-bit word to 32-bits, sra_32 represents the arithmetic right shift of a 32-bit word, and truncateu16 truncates a 16-bit word to 16-bits.

[7] The unverified AVX2 implementation in the pq-crystals/kyber repository deploys an alternative optimization that also does not require a conditional subtraction.

Crucially, the above spec given by prior work for the computation performed for each coefficient can be converted to a circuit by the new EasyCrypt machinery. This means that proving the new version of the AVX2 compression algorithm only requires massaging the program to allow conversion to a circuit (unrolling loops, using appropriate bindings, etc.) and calling the **bdep** tactic as

```
bdep 16 10 lane_func_compress10 pcond_true
```

EasyCrypt then automatically recognizes the map structure of the function across all 768 coefficients, confirms that all lanes are equivalent, and finally checks that the first lane is equivalent to the specification circuit. Note that here we delegate the goal to a SAT solver, which is able to discharge the latter goal even though the two circuits are nothing alike.

We ran experiments comparing this lane-oriented analysis with direct SAT translation of the full circuit. As expected, for smaller computations such as the initial Barrett reduction step, the difference is noticeable in the timings but both approaches offer reasonable performance levels. However, as the circuit size grows, a direct translation to SAT is not feasible. For example, the full sequence of operations required for compress takes 30s in a very modest machine, whereas a blunt SAT based approach does not produce a result in reasonable time.

7.3. Faster verified code

We now discuss the performance of our AVX2-optimized implementation of ML-KEM. We compare our new optimized implementation with the previous Jasmin implementation [12], the *official* AVX2 optimized implementation released by the Kyber team and available from the pq-crystals repository, and the formally verified implementation available in libcrux. We obtained the code from pq-crystals[8] from the project's GitHub and only did one modification to the corresponding Makefile: we added the option -no-avx512f to avoid the compiler introducing AVX-512 instructions (which happens when the target CPU supports those instructions) and use the shared libraries produced by the corresponding Makefiles. We performed this change because our focus is AVX2 implementations.

For libcrux, we consulted the project's GitHub and chose the C AVX2 implementation.[9] We notice that the authors of this project provide ready-to-use C implementations alongside the Rust implementations, which are formally verified. Given the nature of our analysis, focused on C and assembly code, we found the C implementations suitable for this analysis. To build the library, we configured the build as *Release*, as recommended by the authors in the available documentation. We wrote a wrapper that copies into and from the structures of libcrux, but we do not include this overhead in the benchmark results.

The benchmarking procedure is as follows. The Intel CPUs are 8700K, 11700K, and 13900K, all with Hyper-Threading and TurboBoost disabled, operating at the base frequency. The compiler used to build the benchmarking binaries, pq-crystals, and libcrux was Clang 18.1.8. Each experiment comprises 10000 valid runs of each algorithm. We focus on the derandomized versions, so randomness sampling does not affect the results. We start by generating 10000 keypair seeds, and then, using the seeds, we generate 10000 keypairs and measure the number of CPU cycles for each execution. We keep the median and follow by generating 10000 ciphertexts / shared secrets (using all 10000 different keypairs) and also keep the median. We do the same for the decapsulation. We repeat this experiment 11 times (to avoid outliers) and report the median of those in this paper. As a final note, all pointers, namely, all inputs and outputs of the evaluated functions, are aligned to 64 bytes to reduce benchmarking noise.

CPU	Implementation	keypair	enc	dec
8700K	[12]	94099	94569	97715
	This Paper	40134	40599	43437
	pq-crystals	39722	39761	46161
	libcrux	58231	62588	66385
11700K	[12]	80150	79942	82242
	This Paper	37458	37798	39970
	pq-crystals	36958	38082	42566
	libcrux	55302	56858	61840
13900K	[12]	67514	67384	77760
	This Paper	34732	35212	43784
	pq-crystals	31448	32090	36064
	libcrux	52774	53502	56798

Table 1. CPU cycles of AVX2 implementations.

Table 1 presents our benchmarking results according to the methodology described above. All implementations from this table correspond to the derandomized versions. In comparison with [12], the previously fastest Jasmin formally verified implementation, we were able to improve performance up to 2.3x on older CPUs, such as the shown Intel 8700K (Coffee Lake), and on more recent ones, such as the Intel 13900K (Raptor Lake) and Intel 11700K (Rocket Lake), between 1.7x and 2.1x.

It is worth noting that the proposed implementation in this paper includes protections against Spectre v1 while the others on the table do not. Spectre v1 has an impact on performance: for instance, on the 11700K, this paper's implementation without Spectre v1 protections is slightly faster, taking 36818, 37200, and 39376 CPU cycles instead of the reported values of 37458, 37798, and 39970. The difference is similar in other CPUs.

Our code compares quite well with the C/asm AVX2 imlementation from pq-crystals: the decapsulation is faster on 8700K and 11700K, comparable for the remaining functions in these CPUs, and somewhat comparable on the 13900K. We expect that the approach and tools defined in this paper can be used to improve performance in specific micro-architectures without significant human effort to fix the proofs of rearranged versions that take advantage, for instance, of different CPU instruction latencies. At this stage, it is also important to highlight that this paper's implementation has the advantage that one does not need to

[8] Link to pq-crystals github.

[9] Link to libcrux github.

rely on Clang or other compilers not breaking the constant-time property.

Regarding the libcrux implementation, it offers interesting performance, considering that it is proven functionally correct at the source level (the C compiler is in the TCB). We note that the chosen implementation is recent and updated frequently, so we expect its performance to improve wrt what we report here. Nevertheless, we expect our code to remain competitive in the analyzed contexts and to provide a formally verified alternative for the analyzed scenarios.

7.4. Easier proofs of vectorized Keccak

To further illustrate the benefits of the paradigm shift we propose, we revisited the correctness proof of the Jasmin re-implementation of openssl's AVX2 implementation, originally presented in [13]. This example exhibits some peculiarities shared by other notable examples of high-speed vectorized implementations, such as a permuted (redundant) state representation benefiting from the available AVX2 instructions and the blurring of the structure of the algorithm into large chunks of straight-line code.

We use as baseline the original proof from [13], which is purely deductive and is spread between 3 files totaling ∼3.5K lines of EasyCrypt code. It must be stressed that this total includes a reusable library (∼750 lines), and that a large percentage of it is based on exhibiting explicit code transformations needed to unlock the equivalence proof. We use circuit-based reasoning to tackle the most challenging part of the proof: the correctness of the round function. The statement reads as follows:

```
M.__keccakf1600_pround_avx2 :
    state = _a ∧ stavx2INV _a ⇒  res = stavx2_keccak_pround _a.
```

where stavx2INV is a predicate that characterizes the storage of the Keccak state, and stavx2_keccak_pround is a circuit-friendly version of the round function without the ι−step. The proof is trivial with the newly developed **bdep** tactic, since it amounts to the equivalence of two circuits (6 lines of proof script running on a couple of seconds). The correctness of the full permutation function follows immediately from standard Hoare reasoning. The whole development has less than 300 lines of code, where most of them are devoted to adjusting the existing specification in a circuit-friendly variant (which would be avoidable if it had been written in the first place with it in mind).

7.5. Bridging the x86-ARM architecture barrier

We conclude this section with another illustration of proof effort amortization with the compression example. Up to now, to the best of our knowledge, there is no fully formally verified proof of an implementation of ML-KEM in ARM platforms. During the course of this work, the Jasmin backend for ARMv7 has matured, and we have developed an implementation. This code is not yet verified, but it already allows us to answer an important question: will one be able to amortize the effort invested in proving the first implementations [11]?

Since the code we have developed for ARMv7 mimics the reference implementation very closely in terms of structure, our goal for future work is to carry out a proof by pure extensional equality, i.e., we prove that all functions provide the same outputs when fed with the same inputs. The relational Hoare logic of EasyCrypt will allow dealing with high-level functions that just call other functions in a reasonably straightforward way, but this will not be the case for low level functions that carry out architecture-specific computations.

We showcase our use of circuit-based reasoning to prove that two different ARM implementations of the compression routines are equivalent to the reference x86-64 implementation. Indeed, in ARMv7 we have several interesting multiplication instructions, which allow different strategies for compression. Our first implementation mimmics the x86-64 reference implementation and uses standard unsigned multiplication `UMULL`. However, the digital signal processing extension implements `SMMULR`, which maps two signed 32-bit integers a and b to $\lfloor ab/2^{32} \rceil$. Choosing $\mathtt{R} = 2^{42}$, we obtain our second implementation of $\lfloor a2^{10}/q \rceil$ as $\lfloor a \lfloor 2^{42}/q \rceil / 2^{32} \rceil$. See Figures 8 and 9 for the ARM implementations. The proofs are identical for both implementations. We prove a relational Hoare triple that states extensional equivalence between the ARM implementations and the x86-64 reference implementation. Note that this equivalence relates a program targeting a 32-bit architecture to a program targeting a 64-bit architecture with totally different instruction sets. The proofs massage both implementations such that they can be converted into circuit form, i.e., unrolling loops and using the correct bindings, and then call the variant of the **bdep** tactic that proves extensional equivalence over the relevant input range described by a predicate pcond_reduced, and with lane structure of the map computation indicated as follows.

```
bdepeq 16 [a] [a] {10 : [rp ∼ rp]} pcond_reduced.
```

Note that, in this case, we do not even require the specifications of the computations carried out by each lane.

8. Related work

One of the most successful applications of formal methods is arguably hardware verification. Two common goals of hardware verification are to prove that a circuit correctly implements a boolean function, or that two boolean circuits are equal [17]. However, there is also a large body of work that targets correctness and equivalence of arithmetic circuits [21]. Our algorithms for converting programs into circuits and for verifying properties of circuits are largely similar to pre-existing algorithms. However, our methodology and the focus on cryptographic implementations appear to be new.

There is a large body of work that uses program verification for proving correctness of cryptographic code [3]. A majority of this work, including [22], [23], [24] and prior work on Jasmin, follows the classic approach of deductive

```
fn compress10(reg ptr u8[
    MLKEM_POLYVECCOMPRESSEDBYTES] rp,
     stack u16[MLKEM_VECN] a) → reg ptr u8[
    MLKEM_POLYVECCOMPRESSEDBYTES]
{
  stack u16[MLKEM_VECN] aa;
  reg u32 c, b, limit, i, j;
  reg u32[4] t; inline int k;

  i = 0; j = 0;
  aa = __polyvec_csubq(a);
  limit = MLKEM_VECN - 3;
  while (i < limit) {
    for k = 0 to 4 {
      t[k] = (32u)aa[(int) i];
      i += 1;
      t[k] <<= 10; t[k] += 1665;
      c = 1290167; c,t[k] = #UMULL(c,t[k]);
      c = 0x3ff; t[k] &= c; }
    c = t[0]; c &= 0xff;
    rp[(int) j] = (8u)c; j += 1;
    b = t[0]; b >>= 8; c = t[1]; c <<= 2;
    c |= b; rp[(int) j] = (8u)c; j += 1;
    b = t[1]; b >>= 6; c = t[2]; c <<= 4;
    c |= b; rp[(int) j] = (8u)c; j += 1;
    b = t[2]; b >>= 4; c = t[3]; c <<= 6;
    c |= b; rp[(int) j] = (8u)c; j += 1;
    t[3] >>= 2; rp[(int) j] = (8u)t[3];
    j += 1; }
  return rp;
}
```

Figure 8. Compress$_{10}$ in ARM 32-bit architecture (reference).

```
fn compress10x4(reg u32 a0 a1 a2 a3 b)
  → reg u32, reg u32 {
  a0=#SMMULR(a0, b);   a1=#SMMULR(a1, b);
  a2=#SMMULR(a2, b);   a3=#SMMULR(a3, b);
  a0=#UBFX(a0, 0, 10); a1=#UBFX(a1, 0, 10);
  a2=#UBFX(a2, 0, 10); a3=#UBFX(a3, 0, 10);
  a0|= a1 << 10; a0 |= a2 << 20;
  a0 |= a3 << 30; a3 >>= 2;
  return a0, a3;
}

fn compress10(reg ptr u8[960] des, stack u16[
    KYBER_VECN] src) →
        reg ptr u8[960] {
  reg u32[4] a; reg u32 b; inline int i j;
  b = #MOV(1321131424 % 65536);
  b = #MOVT(b, 1321131424 / 65536);
  for i = 0 to KYBER_VECN / 4 {
    for j = 0 to 4 {a[j]=(32s)src[4*i+j];}
    a[0],a[3]=compress10x4(a[0],a[1],a[2],a[3],
     b);
    des.[u32 5 * i] = a[0];
    des.[u8 5 * i + 4] = (8u)a[3]; }
  return des;
}
```

Figure 9. Compress$_{10}$ in ARM 32-bit architecture (alternative).

verification, where most of the reasoning is done over the semantics of the target programming language. Our approach enriches program interpretation with an alternative low-level semantics, in order to make the logical structure of computations explicit. We note that our approach is useful for data wrangling and cryptographic operations that do not have a rich algebraic structures, and that other approaches are more suitable when abstract algebraic reasoning is required. In particular, we believe that the approach of CryptoLine [25] can be seen as a hybrid representation that is precisely geared towards algebraic reasoning. However, because of this specific goal, it is less suitable for the class of routines we focus on in this paper. The design of CryptoLine [25] also features automation over scalability and compositionality, so an interesting line for future work is to enable this kind of hybrid reasoning in EasyCrypt as well.

An alternative is to use equivalence checking directly. This is for instance the approach followed by Galois with their Software Analysis Workbench (SAW) [26], which has been used extensively to prove equivalence between a cryptographic implementation and a representation in the Cryptol language [27]. However, traditional equivalence checking suffers scalability issues, as discussed in the body of the paper, and our hybrid approach can be used to improve them. There is also a large body of work on software equivalence checking [28] but these tools are not immediately applicable to cryptographic implementations.

Another traditional approach is to generate correct programs from specifications. This is the approach followed by FIAT Crypto [29]; recently the approach has been extended with mechanisms to synthesize correct and optimized implementations [30]. This approach is appealing but is not applicable to the complex implementations that we consider.

9. Conclusion

We introduce a hybrid representation of programs and an associated verification approach that achieves automation and robustness, which are key benefits of equivalence checking, as well as compositionality and generality, which are key benefits of deductive verification.

The main conceptual insight of the paper is that exposing the high-level logic of the code is key to leverage mainstream circuit-based automated verification. We show that exposing the high-level logic of the code:

1) Can be achieved by moving away from the usual CPU (memory and execution) models and interleaving circuit-based verification with deductive verification (for compositional, incremental/iterative, and scalable reasoning);
2) Allows to verify highly optimized code with reasonable effort, while keeping the complexity of the circuit-based reasoning problem instances under control.

Our experiments highlight that our approach yields significant decreases in (human and computational) verification effort, and potential for proof reuse (across implementations for different platforms as well as evolution of concrete implementations). Indeed, our technical contributions support this conceptual insight and they go beyond the demonstrated verified code: most notably, we define a set of proof-goal patterns (corresponding to new EasyCrypt tactics described in Sections 4 and 5 for automatic circuit-based reasoning that we believe will guide future EasyCrypt functional correctness proofs and may be of independent interest for other formal verification frameworks. Our immediate goal is to leverage this approach to verify more high-assurance cryptographic code, focusing on the recently published NIST post-quantum standards. In particular, our work in the near future will focus on upcoming ML-KEM and MSL-DSA implementations for Armv7 and RISC-V, as they become available, leveraging cross-architecture verification to amortize the overall proof effort.

Acknowledgements

The research was supported by Deutsche Forschungsgemeinschaft (DFG, German research Foundation) as part of the Excellence Strategy of the German Federal and State Governments – EXC 2092 CASA - 390781972 and by the German Federal Ministry of Education and Research (BMBF) in the framework of the 6GEM research hub under grant number 16KISK038.

References

[1] E. W. Smith and D. L. Dill, "Automatic formal verification of block cipher implementations," in *Formal Methods in Computer-Aided Design, FMCAD 2008, 2008*, A. Cimatti and R. B. Jones, Eds. IEEE, 2008, pp. 1–7. [Online]. Available: https://doi.org/10.1109/FMCAD.2008.ECP.10 1

[2] L. Lai, J. Liu, X. Shi, M. Tsai, B. Wang, and B. Yang, "Automatic verification of cryptographic block function implementations with logical equivalence checking," in *Computer Security - ESORICS 2024 - 29th European Symposium on Research in Computer Security, 2024, Proceedings, Part IV*, ser. Lecture Notes in Computer Science, J. García-Alfaro, R. Kozik, M. Choras, and S. K. Katsikas, Eds., vol. 14985. Springer, 2024, pp. 377–395. [Online]. Available: https://doi.org/10.1007/978-3-031-70903-6_19 1

[3] M. Barbosa, G. Barthe, K. Bhargavan, B. Blanchet, C. Cremers, K. Liao, and B. Parno, "SoK: Computer-aided cryptography," in *2021 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2021, pp. 777–795. 1, 16

[4] G. Barthe, B. Grégoire, S. Heraud, and S. Zanella Béguelin, "Computer-aided security proofs for the working cryptographer," in *CRYPTO 2011*, ser. LNCS, P. Rogaway, Ed., vol. 6841. Springer, Berlin, Heidelberg, Aug. 2011, pp. 71–90. 2, 5

[5] G. Barthe, F. Dupressoir, B. Grégoire, C. Kunz, B. Schmidt, and P. Strub, "Easycrypt: A tutorial," in *Foundations of Security Analysis and Design VII - FOSAD 2012/2013 Tutorial Lectures*, ser. Lecture Notes in Computer Science, A. Aldini, J. López, and F. Martinelli, Eds., vol. 8604. Springer, 2013, pp. 146–166. [Online]. Available: https://doi.org/10.1007/978-3-319-10082-1_6 2

[6] J. B. Almeida, M. Barbosa, G. Barthe, A. Blot, B. Grégoire, V. Laporte, T. Oliveira, H. Pacheco, B. Schmidt, and P.-Y. Strub, "Jasmin: High-assurance and high-speed cryptography," in *ACM CCS 2017*, B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, Eds. ACM Press, Oct. / Nov. 2017, pp. 1807–1823. 2, 6

[7] J. B. Almeida, M. Barbosa, G. Barthe, B. Grégoire, A. Koutsos, V. Laporte, T. Oliveira, and P.-Y. Strub, "The last mile: High-assurance and high-speed cryptographic implementations," in *2020 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2020, pp. 965–982. 2, 6

[8] J. W. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehlé, "CRYSTALS - kyber: A cca-secure module-lattice-based KEM," in *2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018*. IEEE, 2018, pp. 353–367. [Online]. Available: https://doi.org/10.1109/EuroSP.2018.00032 2

[9] R. Avanzi, J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehlé, "CRYSTALS-KYBER: Algorithm specifications and supporting documentation (version 3.02)," Round-3 submission to the NIST PQC standardization project, 2021. 2, 3

[10] National Institute of Standards and Technology, "FIPS 203 – module-lattice-based key-encapsulation mechanism standard," 2024. [Online]. Available: https://doi.org/10.6028/NIST.FIPS.203 2, 3

[11] J. B. Almeida, M. Barbosa, G. Barthe, B. Grégoire, V. Laporte, J.-C. Léchenet, T. Oliveira, H. Pacheco, M. Quaresma, P. Schwabe, A. Séré, and P.-Y. Strub, "Formally verifying Kyber episode IV: Implementation correctness," *IACR TCHES*, vol. 2023, no. 3, pp. 164–193, 2023. 2, 3, 10, 11, 13, 14, 16

[12] J. B. Almeida, S. A. Olmos, M. Barbosa, G. Barthe, F. Dupressoir, B. Grégoire, V. Laporte, J.-C. Léchenet, C. Low, T. Oliveira, H. Pacheco, M. Quaresma, P. Schwabe, and P.-Y. Strub, "Formally verifying kyber - episode V: Machine-checked IND-CCA security and correctness of ML-KEM in EasyCrypt," in *CRYPTO 2024, Part II*, ser. LNCS, L. Reyzin and D. Stebila, Eds., vol. 14921. Springer, Cham, Aug. 2024, pp. 384–421. 2, 3, 10, 11, 13, 15

[13] J. B. Almeida, C. Baritel-Ruet, M. Barbosa, G. Barthe, F. Dupressoir, B. Grégoire, V. Laporte, T. Oliveira, A. Stoughton, and P.-Y. Strub, "Machine-checked proofs for cryptographic standards: Indifferentiability of sponge and secure high-assurance implementations of SHA-3," in *ACM CCS 2019*, L. Cavallaro, J. Kinder, X. Wang, and J. Katz, Eds. ACM Press, Nov. 2019, pp. 1607–1622. 2, 16

[14] S. Gueron and F. Schlieker, "Speeding up R-LWE post-quantum key exchange," Cryptology ePrint Archive, Report 2016/467, 2016. [Online]. Available: https://eprint.iacr.org/2016/467 3

[15] E. Alkim, L. Ducas, T. Pöppelmann, and P. Schwabe, "Post-quantum key exchange - A new hope," in *USENIX Security 2016*, T. Holz and S. Savage, Eds. USENIX Association, Aug. 2016, pp. 327–343. 3

[16] J.-C. Filliâtre and S. Conchon, "Type-safe modular hash-consing," in *Proceedings of the 2006 Workshop on ML*, ser. ML '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 12–19. [Online]. Available: https://doi.org/10.1145/1159876.1159880 10

[17] R. K. Brayton and A. Mishchenko, "ABC: an academic industrial-strength verification tool," in *Computer Aided Verification, 22nd International Conference, CAV 2010, 2010. Proceedings*, ser. Lecture Notes in Computer Science, T. Touili, B. Cook, and P. B. Jackson, Eds., vol. 6174. Springer, 2010, pp. 24–40. [Online]. Available: https://doi.org/10.1007/978-3-642-14295-6_5 12, 16

[18] A. Mishchenko, R. Jiang, S. Chatterjee, and R. Brayton, "FRAIGs: Functionally reduced AND-INV graphs," Department of EECS, University of California, Berkeley, Tech. Rep., 2004. 12

[19] C. Barrett, P. Fontaine, and C. Tinelli, "The SMT-LIB standard," Tech. Rep., 2025. 12

[20] H. Becker, V. Hwang, M. J. Kannwischer, B.-Y. Yang, and S.-Y. Yang, "Neon NTT: Faster Dilithium, Kyber, and Saber on Cortex-A72 and Apple M1," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2022, no. 1, pp. 221–244, 2021. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/9295 14

[21] D. Stoffel and W. Kunz, "Equivalence checking of arithmetic circuits on the arithmetic bit level," *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, vol. 23, no. 5, pp. 586–597, 2004. [Online]. Available: https://doi.org/10.1109/TCAD.2004.826548 16

[22] J. K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche, "HACL*: A verified modern cryptographic library," in *ACM CCS 2017*, B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, Eds. ACM Press, Oct. / Nov. 2017, pp. 1789–1806. 16

[23] Y. Zhou, S. Gibson, S. Cai, M. Winchell, and B. Parno, "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*, W. Meng, C. D. Jensen, C. Cremers, and E. Kirda, Eds. ACM, 2023, pp. 2113–2127. [Online]. Available: https://doi.org/10.1145/3576915.3616603 16

[24] A. W. Appel, "Verification of a cryptographic primitive: SHA-256," *ACM Trans. Program. Lang. Syst.*, vol. 37, no. 2, pp. 7:1–7:31, 2015. [Online]. Available: https://doi.org/10.1145/2701415 16

[25] M.-H. Tsai, B.-Y. Wang, and B.-Y. Yang, "Certified verification of algebraic properties on low-level mathematical constructs in cryptographic programs," in *ACM CCS 2017*, B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, Eds. ACM Press, Oct. / Nov. 2017, pp. 1973–1987. 17

[26] R. Dockins, A. Foltzer, J. Hendrix, B. Huffman, D. McNamee, and A. Tomb, "Constructing semantic models of programs with the software analysis workbench," in *Verified Software. Theories, Tools, and Experiments - 8th International Conference, VSTTE 2016, 2016, Revised Selected Papers*, ser. Lecture Notes in Computer Science, S. Blazy and M. Chechik, Eds., vol. 9971, 2016, pp. 56–72. [Online]. Available: https://doi.org/10.1007/978-3-319-48869-1_5 17

[27] I. Galois, *Cryptol: The Language of Cryptography*, 2020. [Online]. Available: https://www.cryptol.net/files/ProgrammingCryptol.pdf 17

[28] B. R. Churchill, O. Padon, R. Sharma, and A. Aiken, "Semantic program alignment for equivalence checking," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, 2019*, K. S. McKinley and K. Fisher, Eds. ACM, 2019, pp. 1027–1040. [Online]. Available: https://doi.org/10.1145/3314221.3314596 17

[29] A. Erbsen, J. Philipoom, J. Gross, R. Sloan, and A. Chlipala, "Simple high-level code for cryptographic arithmetic - with proofs, without compromises," in *2019 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2019, pp. 1202–1219. 17

[30] J. Kuepper, A. Erbsen, J. Gross, O. Conoly, C. Sun, S. Tian, D. Wu, A. Chlipala, C. Chuengsatiansup, D. Genkin, M. Wagner, and Y. Yarom, "Cryptopt: Verified compilation with randomized program search for cryptographic primitives," *Proc. ACM Program. Lang.*, vol. 7, no. PLDI, pp. 1268–1292, 2023. [Online]. Available: https://doi.org/10.1145/3591272 17