The Last Mile: High-Assurance and High-Speed Cryptographic Implementations

José Bacelar Almeida*, Manuel Barbosa†, Gilles Barthe‡, Benjamin Grégoire§
Adrien Koutsos¶, Vincent Laporte§,Tiago Oliveira†, Pierre-Yves Strub‖
*University of Minho and INESC TEC
†University of Porto (FCUP) and INESC TEC
‡MPI for Security and Privacy and IMDEA Software
§Inria
¶LSV, CNRS, ENS Paris-Saclay
‖Ecole Polytechnique

***Abstract*—We develop a new approach for building cryptographic implementations. Our approach goes the last mile and delivers assembly code that is provably functionally correct, protected against side-channels, and as efficient as hand-written assembly. We illustrate our approach using ChaCha20-Poly1305, one of the two ciphersuites recommended in TLS 1.3, and deliver formally verified vectorized implementations which outperform the fastest non-verified code.**

We realize our approach by combining the Jasmin framework, which offers in a single language features of high-level and low-level programming, and the EasyCrypt proof assistant, which offers a versatile verification infrastructure that supports proofs of functional correctness and equivalence checking. Neither of these tools had been used for functional correctness before. Taken together, these infrastructures empower programmers to develop efficient and verified implementations by "game hopping", starting from reference implementations that are proved functionally correct against a specification, and gradually introducing program optimizations that are proved correct by equivalence checking.

We also make several contributions of independent interest, including a new and extensible verified compiler for Jasmin, with a richer memory model and support for vectorized instructions, and a new embedding of Jasmin in EasyCrypt.

1. Introduction

Vulnerabilities in cryptographic libraries are challenging to detect and/or eliminate using approaches based on testing or fuzzing. This has motivated the use of formal verification for proving functional correctness and side-channel protection for modern cryptographic libraries [17], [21], [22], [36]. These approaches have been very successful, to the extent that some of these verified libraries have been deployed in popular products like Mozilla Firefox, Google Chrome, Android, etc. In this paper, we go the last mile: we build and leverage a framework for developing high-assurance and high-speed cryptographic implementations. Our framework delivers efficient, functionally correct, and timing side-channel resistant (vectorized) implementations, without the need to trust the compiler.

Methodology. Our methodology allows developers to follow the typical optimization process for low-level code. We start from a readable reference implementation, for which we prove functional correctness. We then gradually transform the reference implementation into an optimized (possibly platform-specific and vectorized) implementation, and prove that each transformation preserves functional correctness. This approach is similar to the "game-hopping" technique used in provable security, except that we use it for functional correctness of implementations rather than security of high-level algorithms. In parallel, we check that implementations are safe using static analysis techniques, as the previous proofs are carried out in a simpler semantics, which assumes that programs are safe, and compiler correctness is generally stated for safe programs. As a final step, we prove that that the optimized implementations correctly deploy mitigation against timing attacks: we adopt the cryptographic constant-time approach [3], and prove that both the control flow and the memory accesses performed by the optimized program are independent of secret values.

This simple approach has important conceptual benefits. It emulates the developer's mental process for writing optimized implementations, and imposes a convenient separation of concerns between optimization and verification. It also minimizes and helps structuring verification work: functional correctness of the reference implementation is established once and for all, even if the reference implementation is used to derive several platform-specific implementations. Moreover, correctness of transformations can be factored out using generic lemmas. Finally, this approach is compatible with approaches for proving concrete security of cryptographic constructions.

Software infrastructure. We realize our methodology using Jasmin [2], a language and compiler for high-assurance and high-speed cryptography, and EasyCrypt [10], [11], a proof assistant for provable security. The Jasmin language is designed to support "assembly in the head" programming,

i.e. it smoothly combines high-level (structured control-flow, variables, etc.) and low-level (assembly instructions, flag manipulation, etc.) constructs. This combination makes it possible to program by "game-hopping". The EasyCrypt proof assistant supports program logics for reasoning about correctness and equivalence of imperative programs. It has been used to mechanize "game-hopping" security proofs for many cryptographic schemes. Neither Jasmin nor EasyCrypt has been used previously for proving functional correctness of implementations. However, taken together, they provide a convenient framework to develop efficient verified implementations by "game-hopping".

We formally verify Jasmin implementations. These implementations are *predictably* transformed into assembly programs by the Jasmin compiler. Predictability empowers Jasmin programmers to develop optimized implementations with essentially the same level of control as if they were using assembly or domain-specific languages such as qhasm. This means, in particular, that one can express in Jasmin the same optimizations performed in assembly, but using a language with higher-level constructs that favour verification. Moreover the compiler is verified (in the Coq proof assistant) thus guarantees are carried to assembly code.

Technical contributions. We build on Jasmin and EasyCrypt to instantiate a new methodology for the development of high-speed and high-assurance crypto code and demonstrate the resulting framework by giving new, fully verified, assembly implementations of standard cryptographic algorithms that are faster than their best known (non-verified) counterparts. In detail:

1) We enhance the Jasmin framework with a richer memory model, supporting values of different sizes, several language extensions, including intrinsics for vectorized instructions, and a new compiler design that favors extensibility (the proof of compiler correctness has been extended accordingly, as discussed in Section 3.3);
2) We obtain new, highly optimized vectorized implementations for ChaCha20, Poly1305 and Gimli. Our implementations leverage the flexibility of Jasmin, the enhancements presented in this paper and our "game-hopping" approach, to explore the combination of various low-level optimizations to maximize performance benefits;[1]
3) We implement an embedding of Jasmin programs in the EasyCrypt proof assistant [10]. The embedding naturally supports proofs of functional equivalence and functional correctness. We also develop a variant of the embedding to support automatic proofs of protection against side-channel attacks, concretely that control flow and memory accesses are independent from secret inputs (aka. cryptographic constant-time);
4) We prove functional correctness of reference implementations, and equivalence between reference and optimized implementations for ChaCha20, Poly1305 and Gimli. Statements of functional correctness for the first two primitives are taken from, or given in a style similar to, HACL⋆, to guarantee formal interoperability. In the case of Gimli we show how to use a readable Jasmin reference implementation, which can be syntactically very close to the specifications cryptographic standards, as a goal for proving functional correctness of optimized code.

1. While this could have been done in other frameworks, e.g. Vale, we believe that Jasmin is unique in empowering the programmer to fine-tune verification-friendly implementations.

We note that a crucial factor in achieving these results is our ability to tame the verification effort by relying on equivalence checking. Due to the characteristics of Jasmin, the verification workload for the reference implementations is comparable (or even smaller, due to the less elaborate memory model) to that of carrying out formal verification of C code. The power of the relational reasoning offered by EasyCrypt permits bridging reference implementations and optimized implementations with relatively low effort, and the automation offered by the tool suffices to deal with proof goals for side-channel protection.

Related Work. Appel and collaborators prove functional correctness of C implementations of SHA-256 [7], HMAC [14] and HMAC-DBRG [35]. Their proofs are carried in the Verified Software Toolchain [6], an interactive program verification tool for C programs. Their verified implementations can be compiled to assembly using CompCert [26], a formally verified optimizing compiler for C. Their work does not analyze the penalty of using a verified compiler, nor does it provide any guarantee with respect to side-channels.

HACL⋆ [36] is a portable C library that implements many modern cryptographic primitives. Implementations are written in F⋆ [32], a SMT-based verification-oriented language, and formally verified against a readable mathematical specification. By enforcing that secrets are used parametrically, it is also possible to guarantee that F⋆ programs are protected against side channels in the cryptographic constant-time model [3]. Verified F⋆ implementations are first compiled into C using Kremlin, a highly optimized compiler from F⋆ to C [30], and to assembly, using the Clang compiler or the CompCert compiler. Their library is deployed in Mozilla Firefox, Wireguard and other popular products. Their evaluation shows their libraries to be as fast as unverified C libraries, but for this one must assume that functional correctness and protection against timing attacks is preserved by compilation. In contrast, Jasmin does not include a C compiler in its Trusted Computing Base.

Vale [17] supports formal proofs of functional correctness and side-channel protection of assembly-level cryptographic implementations. Functional correctness is proved using the Dafny verifier: implementations are annotated with assertions and checked using verification condition generation and SMT solvers. Vale and Jasmin are similar in the sense that they can verify hand-tuned implementations, without the need to trust a compiler. Vale is able to verify off-the-shelf implementations. In contrast, Jasmin is conceived as a framework (language, compiler, and verification infrastructure) for building new implementations. In particular, the Jasmin language contains high-level constructs that

ease programming, optimization and verification, and the EasyCrypt back-end provides support for proving functional equivalence and reductionist arguments. In short, while the assembly code that can be verified by Vale and Jasmin is essentially the same (see Section 6), the methodology and tool support to obtain this code are different.

The Vale/F* framework [22] builds on Vale to develop an approach based on F* for proving correctness of C programs with inlined x64 assembly. Their approach is based on defining a deep embedding of x64 assembly and formally verifying an executable verification condition generator for x64 assembly programs. The latter is used in combination with F* verification condition generator for proving correctness proofs of hybrid programs. The Kremlin compiler is then used to generate C programs with inlined assembly, to get a performance similar to Vale. In contrast with Jasmin, this approach simplifies the integration of assembly code with general-purpose code written in the C language. However, their Trusted Computing Base includes a C compiler and the interaction between C and assembly.

Independently, Erbsen *et al.* [21] develop an infrastructure for generating verified C implementations of elliptic curve arithmetic from high-level descriptions written in the Coq proof assistant. In addition, their generated code is protected against side channel attacks in the program counter model [28], since it does not contain any branching statement. FiatCrypto implementations are deployed in Chrome, Android and other popular products. As for HACL*, their verified C implementations can be as efficient as the fastest unverified C implementations, when compiled with a non-verified compiler. In this case, their Trusted Computing Base includes a C compiler.

Yang *et al* [19], [29], [33] develop highly automated tools for proving functional correctness of efficient assembly implementations of elliptic curve cryptography.

Lim and Nagarakatte [27] develop an automated method for proving equivalence of assembly implementations of cryptographic libraries. Their method is able to establish equivalence of scalar and vectorized implementations.

In addition, many works focus exclusively on side-channel resistance and/or provable security. In particular, there exist several tools for proving constant-time security [3], [8], [31] or for making programs constant-time by compilation [18], [34]. Finally, a recent work develops a certified compiler [13] that preserves constant-time security.

Scope of the paper. We demonstrate our approach for selected case studies. We focus on functional correctness, and in particular on the "game-hopping" approach to functional correctness. However, we stress that all of our implementations have also been verified for safety and timing side-channel protection.

Our case studies (ChaCha20 and Poly1305) are limited to the realm of symmetric cryptography and we have not formally verified their concrete provable security. We chose these examples to showcase the benefits of our approach, specially for vectorized implementations. Verifying their provable security is the subject of ongoing work.

Our framework can be used to verify other primitives and connect their provable security to functionally correct efficient implementations, as shown by follow-up work on SHA3 [1]. Furthermore, Poly1305 relies on multi-precision modular arithmetic computations that are similar to those employed in several public-key primitives, so carrying out functional correctness proofs for implementations of, for example, the high-speed Jasmin implementation of Curve25519 in [2] is clearly within reach and is a direction for future work. Finally, our framework currently supports x86 platforms, but we plan to extend it to support ARM platforms.

Software and proofs. https://github.com/tfaoliveira/libjc.

Outline. In the next section we use an example to illustrate our methodology. In Section 3 we describe our extensions to Jasmin and in Section 4 we describe the supporting development in EasyCrypt. Then, in Sections 5 and 6 we describe the ChaCha20 case study and provide a thorough performance evaluation of our code in comparison to alternative implementations. Concluding remarks appear in Section 7. The Gimli case study appears in Appendix B.

2. Motivating example: Poly1305

We illustrate our methodology using Poly1305 [15], an authentication algorithm that is used together with ChaCha20 as one of the two ciphersuites recommended in the TLS 1.3 RFC. Poly1305 is a one-time authenticator (the key should only be used once) that allows the sender to attach a cryptographic tag t to a transmitted message m. The receiver of the message should be able to derive the same session key k autonomously, and recompute the tag on the received message. If the tags match, the receiver is assured that only the sender could have transmitted it, provided k is secret and authentic.

Algorithm overview. Poly1305 takes a 32-byte one-time key k and a message m and it produces a 16-byte tag t. The key k is seen as a pair (r, s), in which each component is treated as a 16-octet little-endian number, with the following format restrictions: octets $r[3]$, $r[7]$, $r[11]$ and $r[15]$ should have their top 4 bits cleared, whereas octets $r[4]$, $r[8]$ and $r[12]$ are required to have their two lower bits cleared. For the purpose of this paper we will assume that $k = (r, s)$ is generated as a pseudorandom 256-bit string, after which r is *clamped* to its correct format.

To authenticate a message m, it is split into 16-byte blocks m_i, for $i \in [1, 2, \dots]$. Each block m_i is then converted into a 129-bit number b_i by reading it as a 16-byte little-endian value and then setting the 129-th bit to one (the last block is treated differently). The authenticator t is computed by sequentially accumulating each such number into an initial state $a_0 = 0$ according to the following formula: $a_i = (a_{i-1} + b_i) \times r \pmod p$, for $i \in [1, 2, \dots]$ and where $p = 2^{130} - 5$ is prime. Finally, the secret key s is added to the accumulator (over the integers) and the tag t is simply the lowest 128 bits of the result serialized in little-endian

```
op poly1305_pre (r : zp) (s : int) (m : Zp_msg)
              (mem : global_mem_t) (inn, inl, kk : int)  =
  let body i = let offset = i * 16 in
    if i < size m − 1
    then load_block mem (inn+offset) 16
    else load_block mem (inn+offset) (inl−offset) in
  let n = ceil (inl / 16) in
  m = [body 0; ···; body (n−1)] ∧
  r = load_clamp mem kk ∧
  s = to_uint (loadW128 mem (kk + 16)).

op poly1305_post mem_pre mem_post outt rr ss mm =
  mem_post =
   storeW128 mem_pre outt (W128.of_int (poly1305_ref rr ss mm)).
```

Figure 1: Poly1305 specification in EasyCrypt.

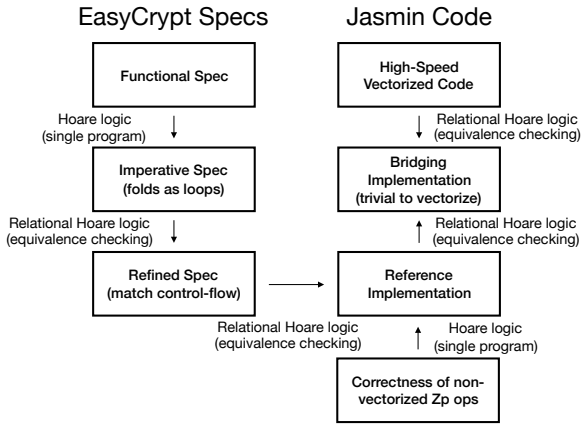

Figure 2: Sequence of hops for Poly1305 correctness proof.

order. The choice of p is crucial for optimization, as it is close to a power of 2: modular reduction can be performed by first reducing modulo 2^{130} and then adjusting the result using a simple computation that depends on the offset 5.

Specification. Our goal is to prove that our optimized implementation of Poly1305 is functionally correct with respect to the high-level specification presented in Figure 1. The specification is written in EasyCrypt and matches the HACL⋆ specification for Poly1305 in that the computation of the tag is expressed as the following functional operators, which iterate over a list of values in $\mathbb{Z}_p$, corresponding to the message blocks and accumulate the state of the authenticator as described above, before finally adding the secret key component s.

```
op poly1305_loop (r : zp) (m : Zp_msg) (n : int) =
  foldl (fun h i ⇒  (h + nth m i) * r)  0_{Z_p}  [0;···;n−1].

op poly1305_ref (r : zp) (s : int) (m : Zp_msg) =
  let h' = poly1305_loop r m (size m) in
  (((asint h') %  2^128 ) + s) %  2^128  .
```

This specification is used to express the following correctness contract over the execution of our implementations:

```
Glob.mem = mem ∧ args = (out,inn,inl,k) ∧
  poly1305_pre r s m mem inn inl k ⇒
      poly1305_post mem Glob.mem out r s m
```

The contract relies on an axiomatic model of the Jasmin language semantics that has been created in EasyCrypt. In this particular case, the contract imposes that the memory Glob.mem in the final state is identical to the initial memory, except for the fact that it now encodes the correct authenticator at position out. Correctness is defined with respect to the values stored in the initial memory, whose contents are interpreted (according to the encoding rules of Poly1305) as containing a message m of length inl bytes stored at position inn, and a key with components r and s stored at position k.

In detail, the precondition and post-condition are shown in Figure 1. The pre-condition requires the implementation to correctly *lift* the message encoded in memory to some representation of $\mathbb{Z}_p$, tweaking the necessary bits as specified by Poly1305. This lifting is defined by operators load_block and load_clamp that specify how the provided memory region is interpreted as a representative of $\mathbb{Z}_p$. We illustrate the latter operator:

```
op load_clamp(mem: global_mem_t) (ptr : address) =
  let x = loadW128 mem ptr in
  let xclamp =
    x & (W128.of_int 0xFFFFFFC0FFFFFFC0FFFFFFC0FFFFFFF) in
  Zp.inzp (W128.to_uint xclamp).
```

The lifting to $\mathbb{Z}_p$ is specified by first interpreting the memory region as a 128-bit integer, applying the bit resets prescribed by Poly1305, and interpreting the resulting value as a residue modulo p (the mapping between integers modulo p and $\mathbb{Z}_p$ is defined by operators inzp and asint). Conversely, the post-condition requires the implementation to correctly encode the final tag (a 128-bit integer) back into memory.

Implementations. We reach our optimized code from the specification through a sequence of implementations, which we depict in Figure 2

abstract implementations: we first create an imperative version of the specification (see Figure 5) that relies on computations over the abstract type zp. We then apply a series of transformations (including loop transformations and code modularization using inlineable functions) to obtain a code that approximates the control flow from Figure 3, and which we explain below. This sequence is shown on the left-hand side of Figure 2.

reference implementations: we replace computations in zp with calls to functions that deal with explicit representations of values in zp. Intuitively, this program corresponds to a Jasmin program, whereas the abstract implementations are just EasyCrypt programs which we use as proof artifacts. However, this reference implementation is not yet fully optimized. This hop is shown on the bottom of Figure 2; each operation of $\mathbb{Z}_p$ must be proved correct in isolation; such results are then used to justify the jump from an abstract implementation to a fully concrete implementation.

optimized implementations: we apply a series of transformations to restructure the reference implementation in a

code that exhibits parallelism and replaces sequential code by vectorized instructions. This sequence of hops is shown on the right-hand side of Figure 2; intuitively we restructure the reference implementation into a new one, for which replacing sequences of instructions with parallel ones is trivial, which then enables the equivalence proof with the fully optimized code.

In what follows, we first give a brief overview of Jasmin, then describe our high-speed implementation of Poly1305, and then explain the various parts of the functional correctness proof.

Background on Jasmin. Jasmin [2] is a language designed for building efficient and formally verified cryptographic primitives within a single language. This entails empowering Jasmin programmers to use different programming idioms for different parts of the implementation, as shown in Figures 4, 9 and 10. In Appendix A we give a by-example overview of the current status of Jasmin.

Jasmin aims to provide the highest level of control and expressiveness to programmers. Informally, the essential property that Jasmin aims to achieve is *predictability*: the expert programmer will be able to precisely anticipate and shape the generated assembly code, so as to be able to achieve optimal efficiency. In this, Jasmin is heavily inspired by qhasm. This means that the programmer must specify the storage for program variables (stack, register) and must handle spilling explicitly (the compiler will fail if it cannot find a spill-free allocation). Jasmin also ensures that side-effects are explicit from the program code by treating flags as boolean variables; this not only gives explicit control over flags, but also makes verification of functional correctness and constant-time security significantly simpler, as all non-memory-related instructions can be treated as pure operators.

On the other hand, Jasmin provides a uniform syntax that unifies machine instructions provided by different micro-architectures. The main purpose of this syntax is to ease programming and to enhance portability.[2] Jasmin also supports (inlineable) function calls, which naturally leads to a style of programming that favors modularity, and supports high-level control-flow structures, rather than jumps. Jasmin also supports functional arrays for describing collections of registers and stack variables. This notation leads to compact and intuitive code and simplifies loop invariants and proofs of functional correctness. Arrays are meant to be resolved at compile-time, and so they can only be indexed by *compile-time expressions*. These can be used to describe statically unrollable for loops and conditional expressions.

These choices have no impact on the efficiency of the generated code, as low-level cryptographic routines usually have a simple control-flow, which is easily captured by these high-level constructions. Moreover, it considerably simplifies verification of functional correctness, safety and side-channel security, and is critical to leverage off-the-shelf verification frameworks, which are often focused on high-level programs.

2. Platform-specific instructions are also available and can be used whenever important, e.g., for efficiency. In particular, programmers may always use a Jasmin dialect where there is a strict one-to-one mapping between Jasmin instructions and assembly instructions.

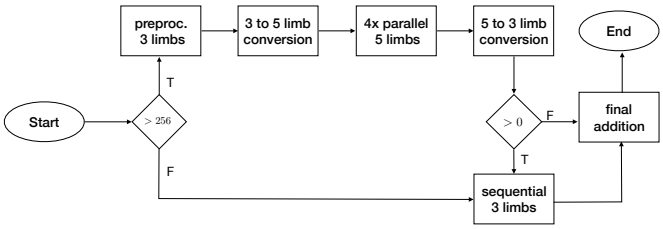

Figure 3: Structure the optimized Poly1305 implementation.

High-speed high-assurance implementation. Our fully optimized code takes advantage of the "assembly in the head" style of programming supported by Jasmin. We rely on the high-level constructs in Jasmin to deploy mixed representation optimizations, which combine sequential and parallel processing as shown in Figure 3.

The implementation first checks whether we are dealing with a small or large message (over 256 bytes). For small messages, it calculates the tag by representing values in $\mathbb{Z}_p$ packed into three 64-bit words (the most significant word for a residue will only use 2 bits). For large messages, a mixed representation computation is used. First, some precomputation necessary for parallel calculations is performed using the packed representation; then the values are converted to a 5-limb representation using radix 2^{26} stored into five 64-bit words. This leaves room in each word so that limb-wise multiplication can be performed safely in 64-bit architectures, as well as accumulating multiple additive carry operations. Parallel computation of 4 message blocks at a time is then implemented using vectorized operations over this representation. Finally, the result is converted back to the packed representation and any remaining message blocks are processed as for short messages.

The high-level control flow and (inlineable) function modularization of Jasmin are crucial to allow managing the code complexity, whereas the low-level features permit controlling instruction selection and scheduling in order to fine-tune performance. An example of our use of the low-level features of the language is given in Figure 4. The optimized implementation relies on AVX2 SIMD instructions, for which Jasmin provides syntactic sugar: shift and add operators are annotated with type information (4u64) indicating that the selected instructions act on 4 unsigned 64-bit words in parallel. Indeed, this code snippet is a part of the parallelized 5-limb implementation, as can be seen by the type of the input x, which contains 4 values in $\mathbb{Z}_p$, each represented using 5-limbs. All of these values are processed in the same way using the SIMD instructions.

Correctness of abstract and reference implementations. The proof of the baseline abstract implementation with respect to the functional specification uses standard Hoare logic. The proof of the remaining abstract implementations is carried by "game hopping", i.e. each step is justified using relational Hoare logic.

```
fn carry_reduce(reg u256[5] x, reg u256 mask26) -> reg u256[5] {
  reg u256[2] z;
  reg u256 t;

  z[0] = x[0] >>4u64 26;      z[1] = x[3] >>4u64 26;
  x[0] &= mask26;             x[3] &= mask26;
  x[1] +4u64= z[0];           x[4] +4u64= z[1];
  z[0] = x[1] >>4u64 26;      z[1] = x[4] >>4u64 26;

  t = z[1] <<4u64 2;
  z[1] +4u64= t;

  x[1] &= mask26;             x[4] &= mask26;
  x[2] +4u64= z[0];           x[0] +4u64= z[1];
  z[0] = x[2] >>4u64 26;      z[1] = x[0] >>4u64 26;
  x[2] &= mask26;             x[0] &= mask26;
  x[3] +4u64= z[0];           x[1] +4u64= z[1];

  z[0] = x[3] >>4u64 26;
  x[3] &= mask26;
  x[4] +4u64= z[0];

  return x;
}
```

Figure 4: Example of optimized Jasmin low-level code.

```
proc poly1305 (out in_0: address, inlen: int, k: address) : unit = {
  var r,h,x: zp;
  var b16: u64;
  var s, h_int : int;

  r ← load_clamp Glob.mem k;
  h ←  0_{Z_p} ;

  while (16 ≤ inlen) {
    x ← load_block Glob.mem in_0 16;
    h ← h + x;     (* Addition in Zp *)
    h ← h * r;     (* Multiplication in Zp *)
    in_0 ← in_0 + 16;
    inlen ← inlen − 16;
  }
  if (0 < inlen) {
    x ← load_block Glob.mem inlen in_0;
    h ← h + x;
    h ← h * r;
  }
  h_int ← (asint h) %  2^128 ;
  s ← W128.to_uint (loadW128 Glob.mem (k + 16));
  h_int ← (h_int + s) %  2^128
;
  Glob.mem ← storeW128 Glob.mem out (W128.of_int h_int);
 }
```

Figure 5: Imperative reference specification in EasyCrypt.

The proof of equivalence from the baseline abstract implementation uses a series of functional correctness lemmas that modularize the computation of each operation in zp in the two representations described above, including arithmetic, conversion and load/store operations.

Figure 6 shows one such auxiliary lemma for the conversion of the results of the 4 parallel computations over

```
lemma add_Rep5_Pack_spec hh1 hh2 hh3 hh4 :
 phoare [ Mhop3.add_Rep5_Pack_Rep3 :
  bRep5 27 hh1 ∧ bRep5 27 hh2 ∧ bRep5 27 hh3 ∧ bRep5 27 hh4 ∧
    hh1 = h1 ∧ hh2 = h2 ∧  hh3 = h3 ∧  hh4 = h4 ⇒
      ubW64 4 res[2] ∧
          repres3 res = repres5 hh1 + repres5 hh2 +
                        repres5 hh3 + repres5 hh4 ] = 1%r.
```

Figure 6: Example of representation correctness lemma. In addition to functional correctness, the contract binds the implementation to using only 4 bits on the most significant output limb, assuming that all the input limbs are using at most 27-bits.

the 5-limb representation to a single value over 3-limb representation. Unlike the rest of the correctness proof, which requires minimal effort, the proofs of these lemmas require ingenuity and user interaction. This is because the proofs of these auxiliary lemmas use algebraic reasoning similar to proofs of other multi-precision computations that are common in cryptography [19]. These proofs are reusable and they could be partially automated (see Sections 4 and 1).

We note that this strategy of performing several hops with abstract implementations considerably simplifies the equivalence proofs.

Equivalence checking the vectorized implementation. At this point in the proof, all the non-vectorized code in the reference implementation matches the code in the extracted Jasmin implementation. The final part of the proof connects our reference implementation to the fully optimized code via three hops that rely on two new dedicated EasyCrypt features:

- The first hop rearranges code so that code corresponding to each single-instruction-multiple-data (SIMD) instruction is modularized as a call to a procedure in a special EasyCrypt module called Ops. Here we take advantage of a new EasyCrypt meta-tactic that permits rearranging n repetitions of the same sequence of k instructions into a sequence of k blocks, each with n identical instructions (intuitively each of these blocks corresponds to a SIMD instruction).
- The second hop replaces calls to Ops with calls to a different module OpsV, where the content of each procedure now makes a single call to the corresponding SIMD instruction (see Figure 7 for an illustrative example). This hop is justified with a once-and-for-all proof that, using the axiomatic semantics of Jasmin expressed in EasyCrypt, establishes the functional equivalence of the Ops and OpsV modules.
- The final hop simply shows that the EasyCrypt implementation obtained from the previous transformation is equivalent to the extracted optimized Jasmin code.

In Section 5 we report on other equivalence proofs we have completed and also on the performance enhancements obtained via the associated optimizations.

```
type vt4u64 = u256.
type t4u64 = u64 array4.

module OpsV = {
  proc iVPBROADCAST_4u64(v : u64) : vt4u64 = {
    return x86_VPBROADCAST_4u64 v;
  } ··· }.

module Ops = {
  proc iVPBROADCAST_4u64(v : u64) : t4u64 = {
    var r : t4u64;
    r[0] ←v;    r[1] ←v;    r[2] ←v;    r[3] ←v;
    return r;
  } ··· }.
```

Figure 7: Ops and OpsV modules.

Protection against timing attacks. Low-level cryptographic software implementations are expected to satisfy a security-critical non-functional property commonly known as *constant-time*, as mitigation against timing attacks. This mitigation consists of the following two restrictions: i. all branching operations depend only on public values (here *public* is defined by explicitly identifying which parts of the initial state can influence the control-flow); and ii. the memory addresses that are accessed by the implementations depend only on public values. Intuitively, these restrictions combined with mild assumptions on the underlying hardware processor guarantee that the execution time of the program (even accounting for micro-architectural features like cache memories) will be fixed once the public part of the input is fixed, ruling out timing attacks.

It is well-known that mitigation against timing attacks can be modeled as observational non-interference. That is, one can define an instrumented semantics, where a distinguished leakage variable (modeled as a list of events) records execution of time-varying instructions, targets of conditional jumps and memory accesses. Then, a program is secure iff every two executions with possibly different secret inputs (but equal public inputs) yield equal leakage.

We define an embedding of the instrumented semantics of Jasmin programs in EasyCrypt and use this embedding to translate our optimized implementation of Poly1305. The proof that a program correctly implements mitigation against timing attacks is carried out with minimal user interaction using existing tactics for relational Hoare logic.[3]

Guarantees over assembly code. The Jasmin compiler is formally verified for functional correctness (in Coq), therefore we know that safe Jasmin source programs are compiled into safe and functionally equivalent assembly programs—we discuss safety in detail in Section 3. In addition, Almeida *et al* [2] informally argue that the Jasmin compiler preserves mitigations against timing attacks in the constant-time model. This informal claim has been reinforced by a formal proof that many optimization passes preserve such mitigations [13], although the connection between these two works has not been established yet.

3. This method for checking constant-time works for a wide range of adversary models, and the translation to EasyCrypt can be easily adjusted to take into account that a given instruction cannot be securely used on secret-dependent data, or only under some restrictions. Considering refined models of leakage, where a richer semantics of instruction is required (e.g., for floating point operations [5] not currently supported by Jasmin) is an interesting direction for future work.

Efficiency. HACL⋆ contains a verified C implementation of Poly1305. Vale [17] and Vale/F* [22] also prove functional correctness of the assembly implementation of Poly1305 for 64-bit operations. In section Section 6 we compare our code with these and other implementations and show that, by fine-tuning it at the Jasmin level, we are able to match and even outperform the best non-verified code.

We note that there is no magical way of improving performance and we do not claim to have a way to replace the expert programmer. Indeed, we apply the same optimization ideas used by the best implementations, namely those adopted in OpenSSL and [24]. What we *do* claim is that we have a tool-supported methodology that allows us to bring the expert programmer on-board in the process of verifying functional correctness, in that many of the hand-crafted optimization steps can be justified with equivalence-checking proofs that are simple to carry out with the developers' input. This includes vectorization and instruction scheduling as prominent examples. Furthermore, the more challenging parts of the functional correctness proofs are not harder (or even simpler due to the memory model of Jasmin) than proofs carried out over high-level languages such as C.

3. Enhancements to Jasmin

In this section we describe the improvements to Jasmin required for writing our highly-optimized fully-verified implementations. This includes language extensions and a significant refactoring of the infrastructure and compiler. Here we start by listing the main challenges we addressed in improving previous work [2].

The original Jasmin framework [2] has been used for writing high-speed cryptographic code. Unfortunately, this framework exhibits three key limitations:

- simplified memory model: only 64-bit values were supported in the original Coq formalization, which excluded implementations that rely, for example, on byte-level access to memory or different views of arrays, and vector instructions;
- monolithic design: the original Coq formalization was developed in the prevailing style for verified compilers, and not intended to be easily extensible;
- basic proof infrastructure: the original implementation featured a verification condition generator (the standard tool for deductive verification) and did not support the process of writing optimized implementations described in the previous section.

We address these issues as follows. First, we develop a new memory model to support value of any width and idioms for efficient low-level code. Second, we propose a

mechanism, called instruction descriptor, which incorporates the information required to handle the instruction from source to assembly, and implement the language and compiler based on instruction descriptors. Third, we develop a verification infrastructure that supports proofs by game hopping. We develop the first two points in this section, and the third point in the next section.

3.1. Compiler design

Compilers, and in particular verified compilers, are typically written for well-defined source languages and architectures. Moreover, it is generally assumed, at least implicitly, that compiler extensions will be developed by compiler writers. This is perfectly reasonable, but has concrete practical implications: any extension or modification to the compiler requires multiple modifications spread over the codebase of the compiler. Furthermore, in the case of verified compilers, it is also necessary to perform multiple modifications across the compiler proof.

In our context, this status quo is unsatisfactory for two reasons: first, the source language is not as closed as traditional languages: in particular, it is designed to support assembly in the head and to grow organically to support richer sets of instructions and eventually multiple platforms. Ideally, these extensions should be manageable by external contributors with limited skills in formal verification. Therefore, we introduce the notion of *instruction descriptor*, which packs all the knowledge and the proof obligations required for adding new instructions to the compiler. The use of descriptors makes the compiler more easily extensible. The approach is generic and applicable to other compilers.

Machine instructions can be made available as *intrinsic* operators at the source level. Due to historical reasons and micro-architectural constraints, each instruction has a specific "calling-convention". For instance, many instructions implicitly write part of their output to the *flags* register. Also, several instructions operate *in place*: their main destination should be the same as one of their sources. These constraints are irrelevant to program verification and enforcing them early in the compilation process would not bring any benefit. As an example, the x86 MUL instruction takes two inputs: one explicitly and the other one implicitly from the RAX register; its output is always written to the RDX, RAX and flags registers. On the opposite, these operators are uniformly described as pure functions at the source level. In the case of the #x86_MUL intrinsic, it takes two values as input and produces seven values as output. Therefore, the compilation of these simple instructions is not trivial: on one hand register allocation must enforce the various architectural constraints; on the other hand the generation of assembly code should check that it is safe to move from a functional semantics to an imperative one with side effects. The associated correctness proof is also tedious and slightly involved.

Therefore, we have built a novel verified compiler infrastructure so that new instruction can be added by constructing a descriptor and adding it to the development. This permits automatically implementing compilation and extending the correctness proof to support the new instruction. In particular, descriptors permit extracting rules for register allocation, as they include meta-data about which registers are read or written by an assembly instruction, which instruction performs memory accesses, etc. This information is also useful to extend support for verification of non-functional properties, in our case mitigation of timing attacks.

For instance, the descriptor for the #x86_MUL sz (non-truncated unsigned multiplication of words of sz bits) is constructed from the following data:

- [:: F OF; F CF; F SF; F PF; F ZF; R RDX; R RAX] the list of destinations where to write the output values, here implicitly to some flag registers and to the RDX and RAX registers;
- [:: R RAX; E sz 0] the list of sources from where to read the input values, here implicitly from the RAX register and from the first argument;
- [:: TYoprd] the type of the emitted instruction, here it has one operand;
- (MUL sz) the instruction to emit.

The descriptor also contains a few well-formedness arguments (proved by computation) and a correctness argument which links the high-level functional semantics of the intrinsic operator to the low-level imperative semantics of the machine instruction.

As an additional advantage of this design, pseudo-instructions can be seamlessly introduced. For instance, a common pattern for zero-initializing a register is to XOR it with itself. However in the Jasmin source language, uninitialized registers have undefined values that should not be used in computations. We have thus added an operator set0 which takes no input and returns a zero value; its descriptor maps it to the $\mathsf{r} \mapsto \mathsf{XOR\ r\ r}$ instruction which takes only one argument: its destination.

3.2. Jasmin language and memory model

Memory model. Although x86_64 microprocessors mainly provide 64-bit registers, programs may manipulate values of various bit widths. Ultimately, there are only registers of 64 bits. Values of smaller sizes do fit, but some bits are undefined. If the value is larger than what is expected by an operation, this is not an issue: only the relevant bits are used. The behavior when writing a small value into a register depends on the operation and its size: the high bits of the destination registers may be preserved or zeroed. The exact behavior at the micro-architectural level is very intricate: it would be unwise to expose it to the programmer. Therefore, we define two semantics: a source semantics that is uniform and convenient for reasoning; and a compiler semantics, in which variables may hold "partial" values, i.e., with some of their most significant bits being undefined.

In any case, operators at some size may be applied to arguments of a larger size: arguments are implicitly truncated. The compiler needs not do anything special, since this is the semantics at the assembly level: operators extract the relevant part from their register operands. Technically, we

```
fn load_add(reg u64[3] h, reg u64 in, reg u64 len) −→ reg u64[3] {
  reg bool cf;
  reg u64 j;
  stack u64[2] s;
  reg u8 c;
  s[0] = 0; s[1] = 0;
  j = 0;
  while (j <u len) {
    c = (u8)[in + j]; s[u8 (int) j] = c; j += 1;
  }
  s[u8 (int) j] = 1;
  cf, h[0] += s[0];
  cf, h[1] += s[1] + cf;
  _, h[2] += 0 + cf;
  return h;
}
```

Figure 8: Load and add the final bytes

require that functions have a signature and every assignment be decorated with the type of the assigned value. By analogy to an identity operator, assignment truncates the value to the given type, which enables to soundly compile copies using a (truncating) mov instructions.

Another extension that permits dealing with varying length operations is a refinement of the memory model specification to allow "type-punning" — reading and writing distinct but overlapping ranges of addresses. Interestingly, the precise behavior need not be specified in order to prove the correctness and security of the compilation. Note however, that to reason about Jasmin programs relying on such memory access patterns, the instance of the memory model might need to be refined.

Flexible views of stack arrays. To efficiently implement functions like the C function *memcpy*, or the processing of the last (potentially incomplete) block of plaintext in a stream cipher, it is important to be able to use the same pointer to read and store data of varying sizes. For memory accesses, this follows a direct consequence of supporting various sizes in Jasmin. However, the stack in Jasmin is not seen as addressable memory at the source level, although stack arrays are compiled as pointers into the stack. To allow the same flexibility in stack operations, we have added a special feature for arrays in the stack, which allows reading/writing words of different sizes, as illustrated in Figure 8.

As can be seen in the figure, a stack array can be seen as a contiguous sequence of bytes, which is very convenient when only a part of the array ends up being used. Aliasing and overlapping accesses issue may thus arise, but they are scoped to a single array: at the source level, stack arrays as a whole enjoy a value semantics, are disjoint, etc.

Vector instructions. Instruction descriptors and our more general memory model allow us to integrate vector instructions in the Jasmin language. Following the design principles of Jasmin, we consider both generic, zero-cost, portable and non-portable instructions.

One example of portable instructions is parallel addition. For these operations, the language provides a convenient

```
fn shuffle_state(reg u256[4] k) −→ reg u256[4] {
  k[1] = #x86_VPSHUFD_256(k[1], (4u2)[ 0, 3, 2, 1]);
  k[2] = #x86_VPSHUFD_256(k[2], (4u2)[ 1, 0, 3, 2]);
  k[3] = #x86_VPSHUFD_256(k[3], (4u2)[ 2, 1, 0, 3]);
  return k;
}
```

Figure 9: Shuffling function of ChaCha20

```
inline fn R4(inline int c, reg u256 x) −→ reg u256 {
  inline int d;
  reg u256 a, b, r;
  global u256 cr, dr;
  cr = c;
  a = #x86_VPSLLV_4u64(x, cr);
  d = (4u64)[64, 64, 64, 64] - c;
  dr = d;
  b = #x86_VPSRLV_4u64(x, dr);
  r = a ^ b;
  return r;
}
```

Figure 10: Parallel rotation function

syntax. For instance, the instruction x +4u64= z; in which variables x and z have type u256, performs four parallel 64-bit additions on vectors x and z and assigns the result to variable x.

Non-portable instructions are available through the general syntax for *intrinsics*, e.g., #x86_VPSHUFD_256. This operator shuffles the four 64-bit elements of its first argument. The exact shuffling is specified by its second argument, which is an 8-bit value. This value is best seen as a vector of four 2-bit numbers describing, for each element of the destination vector, its original position in the source vector. We have introduced a convenient syntax to represent this kind of constant values. For example, Figure 9 shows a shuffling routine that is part of the ChaCha20 operation.

Dynamic globals. The initial version of Jasmin allowed parameters in source code: constants that are inlined very early in the compilation process similarly to C macros. However, not all constants are the same and, for performance reasons, some constants are best stored in the code segment, e.g., to take advantage of RIP-based addressing.[4]

To permit taking advantage of these features, our extension to Jasmin permits tagging local variables as global. These will behave as any other local variable, but will be compiled to a code-segment constant value. For this to be possible, their value should be known at compile-time, after expansion of parameters, function call inlining, loop unrolling and constant propagation. The compiler will ensure that globals with equal values are merged.

This is a very useful mechanism, when an immediate argument to an instruction is best described by a computation, as in vector instructions in which the immediate value describes a permutation, or a vector of shift counts. As an example, the **R4** function shown in Figure 10 performs four parallel rotations on a vector of 64-bit values. The first

4. In this mode, the data is stored within the code segment and referenced through a small offset relative to the current value of the instruction pointer.

argument is a vector of inline values that correspond to the bit counts of these different rotations. These rotations are implemented using one left shift by the given bit count, one right shift by the complementary bit count, and a final XOR of the results of the two shifts. The bit counts are computed as inline values and stored in the code segment.

3.3. Compiler correctness and safety analysis

We have formalized the operational semantics of Jasmin programs and x86 assembly code in the Coq proof assistant. The formalization is based on the new memory model, and supports instruction extensions, including SIMD. We have also developed an extensible compiler architecture based on instruction descriptors, and proved that the compiler is correct. This means that the result of the compilation preserves the semantics of the original Jasmin program, assuming that the program is well-typed, safe, terminating, and accepted by the compiler—the compiler may still fail for well-typed and safe programs, for instance because the compiler does not perform spilling.

We have also extended the Jasmin compiler to verify that the source program is safe, using a fully automated static analyser, as well as terminating, using a simple analysis based on ranking functions. Concretely, for safety we check for the absence of division by zero, out-of-bound array accesses and variable initialization. Moreover, we need to ensure that, during the execution of the Jasmin program, all loads and stores take place in allocated chunks of the memory (i.e. a specification of valid memory regions, which define the memory calling contract). We do not require the user to supply the static analyser with the allocated memory ranges. Instead, we automatically compute an over-approximation of the offsets that must be allocated in the memory. Once the analysis is complete, the user is notified of the inferred ranges, which are sufficient conditions under which the program is safe. Since the offsets accessed in the memory may depend on the inputs of the program, these are symbolic conditions involving the initial value of the inputs. We consider polyhedral conditions, i.e. conjunctions of linear inequalities. For example, in the case of Poly1305, we automatically infer the following ranges:

$$\begin{array}{ll} \mathsf{range(out)}: \ \mathsf{out} + [0; 16[& \mathsf{range(inlen)}: \ \emptyset \\ \mathsf{range(k)} \ : \ \mathsf{k} + [0; 32[& \mathsf{range(in)} \ : \ \mathsf{in} + [0; \mathsf{inlen}[\end{array}$$

Our analysis is based on abstract interpretation techniques [20], and uses the Apron [25] library of numerical domains. To over-approximate the memory accesses, we use a symbolic *points-to* abstraction combined with the polyhedra domain. Operations in the polyhedra domain have a worst-case exponential complexity in the number of variables. Therefore, we perform a pre-analysis to detect which variables must be included in the relational domain. Moreover, we allow to user to help the analysis by indicating which input variables are pointers (k,in and out in Poly1305), and which variables must be included in the relational domain (inlen in Poly1305).

4. Source-level verification

This section describes our embedding of Jasmin in EasyCrypt. We use this embedding for proving correctness of reference implementations, equivalence between reference and optimized implementations, and finally correct mitigation of timing attacks.

4.1. Overview of EasyCrypt

EasyCrypt [10] is a general-purpose proof assistant for proving properties of probabilistic computations with adversarial code. It has been used for proving security of several primitives and protocols [4], [9], [11], [12].

EasyCrypt implements program logics for proving properties of imperative programs. In contrast to common practices (which use shallow or deep embeddings), the language and program logics are hard-coded in EasyCrypt—and thus belong to the Trusted Computing Base. The main program logics of EasyCrypt are Hoare logic, and relational Hoare logics—both operate on probabilistic programs but we only used their deterministic fragments. The relational Hoare logic allows to relate two programs, possibly with very different control flow. In particular, the rule for loops allows to relate loops that do not do the same number of iterations. This is essential for proving correctness of optimizations based on vectorization, or when the optimization depends the input message length.

The program logics are embedded in a higher-order logic which can be used to formalize and reason about mathematical objects used in cryptographic schemes and also to carry meta-reasoning about statements of the program logic. Automation of the ambient logic is achieved using multiple tools, including custom tactics (e.g. to reason about polynomial equalities) and back-end to SMT solvers. For the purpose of this work, we have found it convenient to add support for proof by computation. This tool allows users to perform proofs simply by (automatically) rewriting expressions into canonical forms.

4.2. Design choices and issues

Rather than building a verified verification infrastructure on top of the Coq formalization of the language (a la VST [6]), we opt for embedding Jasmin into EasyCrypt. We choose this route for pragmatic reasons: EasyCrypt already provides infrastructure for functional correctness and relational proofs and achieves reasonable levels of automation. On the other hand, embedding Jasmin in EasyCrypt leads to duplicate work, since we must define an embedding of the Jasmin language into EasyCrypt. Although we already have an encoding of Jasmin into Coq, we cannot reuse this encoding for two reasons: first, we intend to exploit maximally the verification infrastructure of EasyCrypt, so the encoding should be fine-tuned to achieve this goal. Second, the Coq encoding uses dependent types, which are not available in EasyCrypt. However, these are relatively simple issues to resolve, and the amount of duplicate work is largely

compensated by the gains of using EasyCrypt for program verification (also note that building a verified verification infrastructure in Coq requires some effort).

4.3. Embedding Jasmin in EasyCrypt

The native language of EasyCrypt provides control-flow structures that perfectly match those in Jasmin, including if, while and call commands. This leaves us with two issues: 1) to encode the semantics of all x86 instructions (including SIMD) in EasyCrypt; and 2) to encode the memory model of Jasmin in EasyCrypt.

Instruction semantics. Our formalization of x86 instructions aims at being both readable and amenable to building a library of reusable properties over the defined operations, in particular over SIMD instructions. The first step is to define a generic theory for words of size k, with the usual arithmetic and bit-wise operations. The semantics of arithmetic operations are based on two injections (signed and unsigned) into integers and arithmetic modulo 2^k. For bit-wise operations, we rely on an injection to Boolean arrays of size k. Naturally a link between both representations (int and Boolean array) is also created, which allows proving for example that shifting a word $n \ll i$ is the same as multiplying it by $\mathsf{to\_uint}\ 2^i$.

Scalar x86 operations are formalized using the theory for words, and useful lemmas about the semantics of these instructions are also proved as auxiliary lemmas. For example, the formalizations of shl and shr permit proving lemmas like $\mathsf{shl}\ x\ i \oplus \mathsf{shr}\ x\ (k-i) = \mathsf{rol}\ x\ i$, under appropriate conditions on i.

The semantics of SIMD instructions rely on the theories for 128/256 bit words, but the semantics must be further refined to enable viewing words as arrays of sub-words, which may be nested (e.g., instruction vpshufd sees 256-bit words as two 128-bit words, each of them viewed as an array of sub-words). To ease this kind of definition, we have defined a bijection between words and arrays of (sub-)words of various sizes. Then vector instructions are defined in terms of arrays of words.

Memory model. EasyCrypt does not provide the notion of pointer natively. We rely on the concept of a global variable in EasyCrypt, which can be modified by side effects of procedures, to emulate the global memory of Jasmin and the concept of pointer to this memory. A dedicated EasyCrypt library defines abstract type global_mem_t equipped with two basic operations for load mem[p] and store mem[p ← x] of one byte, as follows:

```
type address = int.
type global_mem_t.
op "_[_]" : global_mem_t → address → W8.t.
op "_[_←_]" : global_mem_t → address → W8.t → global_mem_t.
axiom get_setE m x y w : m[x ← w][y] = if y = x then w else m[y].
```

From this basic axiom we build the semantics of load and store instructions for various word sizes. The Jasmin memory library then defines a single global variable Glob.mem of type global_mem_t, which is accessible to other EasyCrypt modules and is used to express pre-conditions and post-conditions on memory states.

Soundness. The embedding of a Jasmin program into EasyCrypt is sound, provided the program is safe. This is because the axiomatic model of Jasmin in EasyCrypt is intended to be verification-friendly, and assuming safety yields much simpler verification conditions and considerably alleviates verification of functional and equivalence properties. This assumption is perfectly fine, since Jasmin programs are automatically checked for safety before being compiled and embedded into EasyCrypt. As potential future work, it would be interesting to make our safety checker certifying, in the sense that it automatically produces a proof of equivalence between the Coq and EasyCrypt semantics of Jasmin programs—technically, this would be achieved by formalizing in Coq a simpler semantics for safe programs, and proving automatically that the two semantics coincide for safe programs. The coincidence between the simpler semantics in Coq and the Jasmin semantics would still need to be argued informally.

Reusable EasyCrypt libraries. In the course of writing correctness proofs for our use cases we have created a few EasyCrypt libraries that will be useful for future projects. In addition to the interchangeability of generic vectorization modules Ops and OpsV which we mentioned in Section 2, significant effort was put into enriching the theories of words in order to facilitate proofs of computations over multi-precision representations. Concretely, a theory was created that permits tight control over the number of used bits within a word (a form of range analysis), which is crucial for dealing with delayed carry operations and establishing algebraic correctness via the absence of overflows. The central part of this library is generic with respect to the number of limbs, so that operations like addition and school-book multiplication can be handled in a fully generic way (here we rely heavily on the powerful ring theory in EasyCrypt). When dealing with constructions such as Poly1305, base on primes which are very close to a power of 2, this means that only the prime-specific modular reduction algorithm needs special treatment. Moreover, this theory was fine-tuned to interact well with SMT provers, enabling the automatic discharge of otherwise tedious to prove intermediate results.

4.4. Verification of timing attack mitigations

The EasyCrypt embedding of Jasmin programs is instrumented with leakage traces that include all branching conditions plus all accessed memory addresses (this also includes array indexes since an access in a *stack* array will generate a memory access at the assembly level). It is then possible to check that the private inputs do not interfere with this leakage trace in the classical sense that, for all public-equivalent input states $x_1 \equiv_{\mathsf{pub}} x_2$, the program will give rise to identical leakages $\ell_1 = \ell_2$. Figure 11 shows an example of the generated instrumented EasyCrypt code.

```
fn store2(reg u64 p, reg u64[2] x) {
  [p + 0] = x[0];
  [p + 8] = x[1];
}
```

```
proc store2 (p:u64, x:u64 array2) : unit = {
  var aux: u64;
  leakages ← LeakAddr [0] :: leakages;
  aux ← x[0];
  leakages ← LeakAddr [to_uint (p + 0)] :: leakages;
  Glob.mem ← storeW64 Glob.mem (to_uint (p + 0)) aux;
  leakages ← LeakAddr [1] :: leakages;
  aux ← x[1];
  leakages ← LeakAddr [to_uint (p + 8)] :: leakages;
  Glob.mem ← storeW64 Glob.mem (to_uint (p + 8)) aux;
}
```

Figure 11: EasyCrypt code (bottom) instrumented for *constant-time* verification of a Jasmin program (top).

Pleasingly, EasyCrypt tactics developed to deal with information flow-like properties handle the particular equivalence relation associated with so-called *constant-time* security extremely effectively. In particular, EasyCrypt provides the sim tactics which is specialized on proving equivalence of programs sharing the same control flow (which is the case here, as we are reasoning about two executions of the same program). The tactic is based on dependency analysis and also proved very useful in justifying simple optimizations like spilling, which do not affect the control flow. In the case of constant-time verification there is a very interesting side-effect to the dependency analysis performed by this tactic: it is able to infer sufficient conditions (equality of input variables) that guarantee equality of output variables. When applied to constant-time verification this means that, when this tactic is successful (which was the case for our use-cases) the user just needs to check if the inferred set of variables are all public. We note that performing this kind of analysis at the assembly level is usually hard. We take advantage of the fact that Jasmin provides a high-level semantics that makes it suitable for verification; in particular, the clear separation between memory, stack variables and stack arrays at source level greatly simplifies the problem.

5. Case Study: ChaCha20

Algorithm overview. ChaCha20 is a stream cipher, which we describe as specified in TLS 1.3. It defines an algorithm that expands a 256-bit key into 2^{96} key streams (each stream is associated with a 96-bit nonce) each consisting of 2^{32} blocks (each 64-byte block is associated with a counter value).[5] ChaCha20 defines a procedure to transform an initial state into a keystream block. The initial state is constructed using the 256-bit key k (seen as eight 32-bit words), the 96-bit nonce n (seen as three 32-bit words), a 32-bit counter b and four 32-bit constants c. Pictorially, the initial state can be seen as the following matrix, where on the left-hand side we show the arrangement of 32-bit words and on the right-hand side we show the matrix entry numbering.

$$\begin{matrix} c & c & c & c \\ k & k & k & k \\ k & k & k & k \\ b & n & n & n \end{matrix} \qquad \begin{matrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{matrix}$$

The state transformation, which is repeated for 10 rounds, is based on the following operation that acts upon four 32-bit words at a time:

$\mathsf{Qround}(a, b, c, d)$:

$$\begin{array}{lll} a \leftarrow a + b; & d \leftarrow d \oplus a; & d \leftarrow \mathsf{rol}\ d\ 16; \\ c \leftarrow c + d; & b \leftarrow b \oplus c; & b \leftarrow \mathsf{rol}\ b\ 12; \\ a \leftarrow a + b; & d \leftarrow d \oplus a; & d \leftarrow \mathsf{rol}\ d\ 8; \\ c \leftarrow c + d; & b \leftarrow b \oplus c; & b \leftarrow \mathsf{rol}\ b\ 7; \end{array}$$

Return (a, b, c, d)

Each round updates the state by gradually modifying the state, four words at a time using the Qround function above, according to the following sequence of 4-word selections: $(0, 4, 8, 12)$, $(1, 5, 9, 13)$, $(2, 6, 10, 14)$, $(3, 7, 11, 15)$, $(0, 5, 10, 15)$, $(1, 6, 11, 12)$, $(2, 7, 8, 13)$ and $(3, 4, 9, 14)$. The final keystream block results from the XOR combination of the output of the 10 rounds with the initial state.

Our implementation. We have defined and proved two versions of ChaCha20, one relying only on scalar operations (no vectorization) and the second one relying on AVX2.

The AVX2 version combines two approaches to the optimization of ChaCha20: for short messages (up to 256 bytes) we follow the lines of [23], whereas for large messages we adopt the strategy of OpenSSL. Both approaches were ported to Jasmin, and further optimization of instruction selection, scheduling and spilling was conducted to obtain additional reductions in cycle counts.

Both approaches rely on vectorized instructions, but with different parallelization approaches. For small messages, two (for messages of up to 128-bytes) or four keystream blocks are computed at a time, as there are enough 256-bit registers available to enable the parallel computation of some steps within the same block using a dedicated state representation.[6] For long messages, this no longer pays off due to the need for spills, and we rely on sixteen 256-bit registers, which permit storing the states for 8 block computations using a direct paralellisation approach that replicates a fast implementation of a single block.

In the next section we give detailed performance benchmarks for our code, and compare to existing implementations. Next, we describe how, in addition to being the fastest, our code is also proved functionally correct.

5. The typical composition with Poly1305, also adopted in TLS 1.3, uses ChaCha20 with counter 0 to generate the key material for Poly1305; the keystream generated for increasing counters starting at 1 is used for encryption by XOR-ing with the plaintext. Poly1305 is then used to authenticate the ciphertext (prefixed with any metadata that must also be authenticated) after adding a length-encoding padding. We analyse the two algorithms in isolation to facilitate comparison with other implementations, and because the verification challenges are significantly different.

6. Four 256-bit registers are used to store two initial states for two successive counters, which permits computing four lines of code in Qround with only three vector instructions, simultaneously for the two states. The round is completed by permuting the states, again using vector instructions, and repeating the same technique to compute the last four lines in Qround.

Formal verification. The scalar and AVX2 versions have (almost) the same specification, which corresponds to the HACL⋆ specification, with some differences we present later. Similarly to what we did for Poly1305, we define an EasyCrypt imperative reference implementation and show that it satisfies HACL⋆ functional specification using Hoare logic. Then, we prove the equivalence between this reference implementation and both of our optimized implementations.

The main challenge when proving correctness of the imperative specification lies in memory operations. The imperative specification stores ciphertext blocks eagerly (512-bits at a time), while the functional specification stores the full ciphertext in one go at the end. Therefore, we need a condition ensuring that stores do not erase the fragment of the initial plaintext that remains to be encrypted. Formally, we require that $\mathsf{plain} + \mathsf{len} \leq \mathsf{output} \vee \mathsf{output} \leq \mathsf{plain}$.

Proving equivalence with the scalar optimized implementation is relatively straightforward. The main difficulties come from optimizations of the memory operations. Indeed, in the optimized version we use 64-bit accesses whenever possible, instead of byte-level accesses as in the reference implementation. This allows to save spilling and to reduce the number of loads and stores by a factor of 8.

The proof of the AVX2 version is more intricate. There are two different implementations for short messages and long messages, but we adopt the same proof strategy in both. We describe the long message case. First we change the control flow of the main loop, so that each loop iteration computes 8 independent states. Then, we lay the groundwork for vectorization: rather than manipulating 8 arrays of sixteen 32-bit words, we now manipulate sixteen arrays of eight 32-bit words (here we leverage EasyCrypt automation significantly). Finally, we prove the we can use AVX2 instructions to replace multiple scalar instructions. Again, the main difficulty is to deal with optimized memory access operations, which now uses 256-bit loads and stores. At this point, the 8 states are represented by a 16×8 matrix, which needs to be transposed in order to be XOR-ed with the plaintext (using 256-bit operations) and stored in memory. For performance reasons, this is done in two steps, each dealing with half of the matrix. Because of this, we need a slightly stronger restriction on the input and output pointers than in the scalar version: they need to be either equal or to point to disjoint memory regions.

6. Benchmarks

Methodology. The performance evaluation of the Jasmin implementations of ChaCha20 and Poly1305 was carried out using the benchmarking infrastructure offered by SUPERCOP, version 20190110. All measurements were performed on an Intel i7-6500U (Skylake) processor clocked at 2.5GHz, with Turbo Boost disabled, running Ubuntu 16.04, kernel release 4.15.0-46-generic. The available compilers for all non-Jasmin code were GCC 8.1 and CompCert 3.4. Unless explicitly stated otherwise, GCC was used.

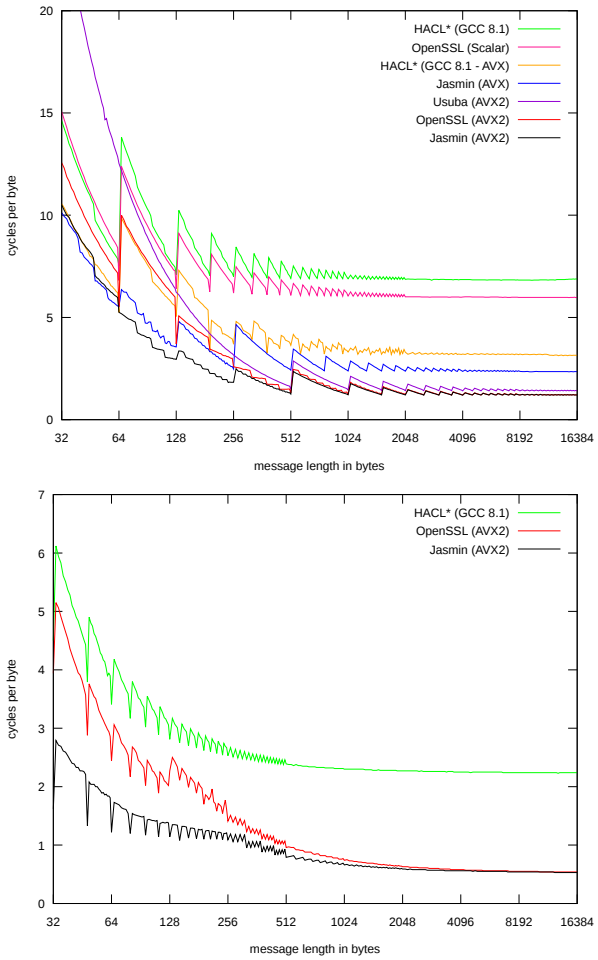

Figure 12: Comparison to non-verified code: ChaCha20 (top), Poly1305 (bottom).

Baselines. Our benchmarks compare the new Jasmin implementations to the fastest implementations for the same primitives and architecture in the following cryptographic libraries: OpenSSL, HACL⋆ and Usuba. We use OpenSSL implementations as references for Vale implementations, given that Vale is able to verify off-the-shelf assembly programs. We integrated external libraries in SUPERCOP by compiling them into static libraries and renaming symbols to remove naming collisions; this is particularly important for libraries which we compiled using different compilers for comparison—for instance HACL⋆ was compiled with both GCC and CompCert. A small patch to the SUPERCOP benchmarking scripts was also added to include these libraries in the set of evaluated implementations. Finally, we created a binding to connect these implementations to the API that SUPERCOP requires for evaluation. Concretely, we implemented APIs crypto_stream_xor for ChaCha20 and crypto_onetimeauth for Poly1305.

Results. Figure 12 shows the benchmarking results of our implementations of ChaCha20 and Poly1305 in comparison to the prominent alternatives in terms of performance. We

emphasize that in this comparison our code is the only one verified for functional correctness, safety and so-called *constant-time* security (HACL⋆ is compiled with non-verifed GCC). The comparison with OpenSSL for small messages should be taken with a grain of salt, as there is some overhead due to binding with SUPERCOP using the C API.

For ChaCha20 the figure shows a clear difference between non-vectorized and vectorized code and our implementation essentially matches OpenSSL as messages grow (we are measuring amortized cycles per byte). In particular note that, for non-vectorized implementations, the C code of HACL⋆ is not much worse than OpenSSL's assembly. The efficiency boost of vectorization is significant, even for relatively small messages. This gives relevance to our results, as we support fully verified vectorized assembly implementations. Note that HACL⋆ includes an AVX implementation of ChaCha20, which we show in the diagram together with our own for the same instruction set, as a representative example of the intrinsic overhead of relying on a C compiler.

For Poly1305 we compare to HACL⋆ and OpenSSL's best implementation, which has a structure similar to ours and uses non-vectorized code for small messages. We can see that our implementation is again the fastest and, more importantly, that vectorized code is once more crucial to make the most of the computational platform (visible for large messages). Interestingly, the figure shows that OpenSSL seems to switch from non-vectorized to vectorized code at around 128-byte messages, whereas our implementation does this at 256-bytes and this seems to be advantageous.

Figure 13 shows a comparison to verified code, where HACL⋆ is now compiled with CompCert. For ChaCha20, we show both our vectorized and non-vectorized implementations, so as to demonstrate that there is indeed a big advantage in bypassing the compiler, even if not relying on vectorization. Indeed, our non-vectorized code is still roughly $\times 2$ faster than HACL⋆, while our vectorized code is about $\times 10$ faster.

For Poly1305 we compare both to HACL⋆ and to non-vectorized OpenSSL code verified in the Vale framework [17] (here the comparison is assembly to assembly and so it is precise). The fine-tuning of our implementation shows in the comparison to the Vale-verified OpenSSL code (the dashed line depicts non-vectorized Jasmin code even for large messages for comparison). The difference to HACL⋆ in this case is huge, both for non-vectorized and vectorized code, and it is due to the intensive use of algebraic operations.

As a final note, we emphasize that we do not claim that ours is the only verification framework that permits achieving such results: for example, the vectorized Poly1305 code from OpenSSL from Figure 12 could be verified using Vale or some other framework and closely match our code's performance (one could also independently verify the assembly code produced by the Jasmin compiler with such tools). The intended take away message from this section is rather that our methodology and framework permit achieving this for *new* implementations, which can incorporate ideas for speed optimization and functional correctness proofs from cryptographers and further fine-tune them using Jasmin.

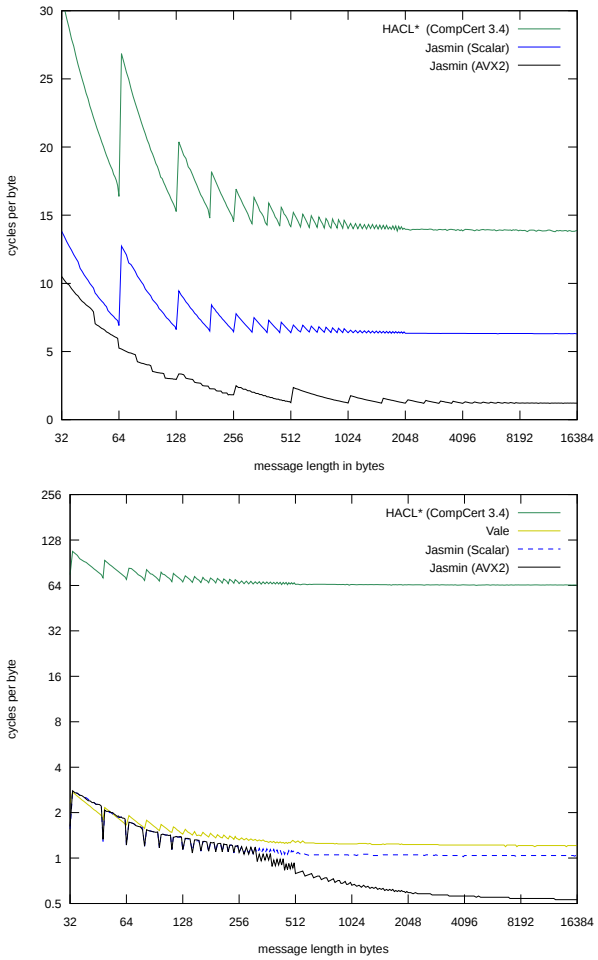

Figure 13: Comparison to verified code: ChaCha20 (top), Poly1305 (bottom).

7. Conclusion

We have developed a practical framework to build high-assurance and high-speed assembly implementations. We have shown the benefits of our approach by manually optimizing and verifying functional correctness and security against timing attacks of code for two primitives from the TLS 1.3. ciphersuite.

There are several important directions for future work. First, we intend to verify a richer set of cryptographic primitives, including all the primitives used in TLS 1.3. Second, we intend to develop a translation validation approach for automating equivalence proofs between reference and vectorized implementations. Third, we intend to extend Jasmin to support other architectures.

Acknowledgements. This work is partially supported by project ONR N00014-19-1-2292. Manuel Barbosa was supported by grant SFRH/BSAB/143018/2018 awarded by FCT. This work was partially funded by national funds via FCT in the context of project PTDC/CCI-INF/31698/2017.

References

[1] J. B. Almeida, Cécile Baritel-Ruet, Manuel Barbosa, Gilles Barthe, Francois Dupressoir, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Alley Stoughton, and Pierre-Yves Strub. Machine-checked proofs for cryptographic standards indifferentiability of sponge and secure high-assurance implementations of sha-3. Manuscript.

[2] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. Jasmin: High-assurance and high-speed cryptography. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 17*, pages 1807–1823. ACM Press, October / November 2017.

[3] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying constant-time implementations. In Thorsten Holz and Stefan Savage, editors, *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, pages 53–70. USENIX Association, 2016.

[4] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, Benjamin Grégoire, Vincent Laporte, and Vitor Pereira. A fast and verified software stack for secure function evaluation. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 1989–2006. ACM, 2017.

[5] Marc Andrysco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. On subnormal floating point and abnormal timing. In *2015 IEEE Symposium on Security and Privacy*, pages 623–639. IEEE Computer Society Press, May 2015.

[6] Andrew W. Appel. *Program Logics - for Certified Compilers*. Cambridge University Press, 2014.

[7] Andrew W. Appel. Verification of a cryptographic primitive: SHA-256. *ACM Trans. Program. Lang. Syst.*, 37(2):7:1–7:31, 2015.

[8] Gilles Barthe, Gustavo Betarte, Juan Diego Campo, Carlos Daniel Luna, and David Pichardie. System-level non-interference for constant-time cryptography. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *ACM CCS 14*, pages 1267–1279. ACM Press, November 2014.

[9] Gilles Barthe, Juan Manuel Crespo, Yassine Lakhnech, and Benedikt Schmidt. Mind the gap: Modular machine-checked proofs of one-round key exchange protocols. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part II*, volume 9057 of *Lecture Notes in Computer Science*, pages 689–718. Springer, 2015.

[10] Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. Easycrypt: A tutorial. In Alessandro Aldini, Javier López, and Fabio Martinelli, editors, *Foundations of Security Analysis and Design VII - FOSAD 2012/2013 Tutorial Lectures*, volume 8604 of *Lecture Notes in Computer Science*, pages 146–166. Springer, 2013.

[11] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. Computer-aided security proofs for the working cryptographer. In Phillip Rogaway, editor, *Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings*, volume 6841 of *Lecture Notes in Computer Science*, pages 71–90. Springer, 2011.

[12] Gilles Barthe, Benjamin Grégoire, Yassine Lakhnech, and Santiago Zanella Béguelin. Beyond provable security verifiable IND-CCA security of OAEP. In Aggelos Kiayias, editor, *Topics in Cryptology - CT-RSA 2011 - The Cryptographers' Track at the RSA Conference 2011, San Francisco, CA, USA, February 14-18, 2011. Proceedings*, volume 6558 of *Lecture Notes in Computer Science*, pages 180–196. Springer, 2011.

[13] Gilles Barthe, Benjamin Grégoire, and Vincent Laporte. Provably secure compilation of side-channel countermeasures: the case of constant-time cryptography. In *Computer Security Foundations*, 2018.

[14] Lennart Beringer, Adam Petcher, Katherine Q. Ye, and Andrew W. Appel. Verified correctness and security of openssl HMAC. In Jaeyeon Jung and Thorsten Holz, editors, *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015.*, pages 207–221. USENIX Association, 2015.

[15] Daniel J. Bernstein. The poly1305-aes message-authentication code. In Henri Gilbert and Helena Handschuh, editors, *Fast Software Encryption: 12th International Workshop, FSE 2005, Paris, France, February 21-23, 2005, Revised Selected Papers*, volume 3557 of *Lecture Notes in Computer Science*, pages 32–49. Springer, 2005.

[16] Daniel J. Bernstein, Stefan Kölbl, Stefan Lucks, Pedro Maat Costa Massolino, Florian Mendel, Kashif Nawaz, Tobias Schneider, Peter Schwabe, François-Xavier Standaert, Yosuke Todo, and Benoît Viguier. Gimli : A cross-platform permutation. In Wieland Fischer and Naofumi Homma, editors, *CHES 2017*, volume 10529 of *LNCS*, pages 299–320. Springer, Heidelberg, September 2017.

[17] Barry Bond, Chris Hawblitzel, Manos Kapritsos, K. Rustan M. Leino, Jacob R. Lorch, Bryan Parno, Ashay Rane, Srinath T. V. Setty, and Laure Thompson. Vale: Verifying high-performance cryptographic assembly code. In Engin Kirda and Thomas Ristenpart, editors, *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017.*, pages 917–934. USENIX Association, 2017.

[18] Sunjay Cauligi, Gary Soeller, Fraser Brown, Brian Johannesmeyer, Yunlu Huang, Ranjit Jhala, and Deian Stefan. Fact: A flexible, constant-time programming language. In *IEEE Cybersecurity Development, SecDev 2017, Cambridge, MA, USA, September 24-26, 2017*, pages 69–76. IEEE Computer Society, 2017.

[19] Yu-Fang Chen, Chang-Hong Hsu, Hsin-Hung Lin, Peter Schwabe, Ming-Hsien Tsai, Bow-Yaw Wang, Bo-Yin Yang, and Shang-Yi Yang. Verifying Curve25519 software. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *ACM CCS 14*, pages 299–309. ACM Press, November 2014.

[20] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252. ACM, 1977.

[21] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. Simple high-level code for cryptographic arithmetic - with proofs, without compromises. In *Proceedings of Security and Privacy 2019*, 2019.

[22] Aymeric Fromherz, Nick Giannarakis, Chris Hawblitzel, Bryan Parno, Aseem Rastogi, and Nikhil Swamy. A verified, efficient embedding of a verifiable assembly language. *PACMPL*, 3(POPL):63:1–63:30, 2019.

[23] Martin Goll and Shay Gueron. Vectorization of ChaCha stream cipher. Cryptology ePrint Archive, Report 2013/759, 2013. http://eprint.iacr.org/2013/759.

[24] Martin Goll and Shay Gueron. Vectorization of poly1305 message authentication code. *12th International Conference on Information Technology - New Generations*, pages 145–150, 2015.

[25] Bertrand Jeannet and Antoine Miné. Apron: A library of numerical abstract domains for static analysis. In *CAV*, volume 5643 of *Lecture Notes in Computer Science*, pages 661–667. Springer, 2009.

[26] Xavier Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006*, pages 42–54. ACM, 2006.

[27] Jay P. Lim and Santosh Nagarakatte. Automatic equivalence checking for assembly implementations of cryptography libraries. In Mahmut Taylan Kandemir, Alexandra Jimborean, and Tipp Moseley, editors, *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2019, Washington, DC, USA, February 16-20, 2019*, pages 37–49. IEEE, 2019.

[28] David Molnar, Matt Piotrowski, David Schultz, and David A. Wagner. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In Dongho Won and Seungjoo Kim, editors, *Information Security and Cryptology - ICISC 2005, 8th International Conference, Seoul, Korea, December 1-2, 2005, Revised Selected Papers*, volume 3935 of *Lecture Notes in Computer Science*, pages 156–168. Springer, 2005.

[29] Andy Polyakov, Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. Verifying arithmetic assembly programs in cryptographic primitives (invited talk). In Sven Schewe and Lijun Zhang, editors, *29th International Conference on Concurrency Theory, CONCUR 2018, September 4-7, 2018, Beijing, China*, volume 118 of *LIPIcs*, pages 4:1–4:16. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018.

[30] Jonathan Protzenko, Jean Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella Béguelin, Antoine Delignat-Lavaud, Catalin Hritcu, Karthikeyan Bhargavan, Cédric Fournet, and Nikhil Swamy. Verified low-level programming embedded in F. *PACMPL*, 1(ICFP):17:1–17:29, 2017.

[31] Bruno Rodrigues, Fernando Magno Quintão Pereira, and Diego F. Aranha. Sparse representation of implicit flows with applications to side-channel detection. In Ayal Zaks and Manuel V. Hermenegildo, editors, *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, pages 110–120. ACM, 2016.

[32] Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. Secure distributed programming with value-dependent types. In Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy, editors, *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*, pages 266–278. ACM, 2011.

[33] Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. Certified verification of algebraic properties on low-level mathematical constructs in cryptographic programs. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 17*, pages 1973–1987. ACM Press, October / November 2017.

[34] Meng Wu, Shengjian Guo, Patrick Schaumont, and Chao Wang. Eliminating timing side-channel leaks using program repair. *CoRR*, abs/1806.02444, 2018.

[35] Katherine Q. Ye, Matthew Green, Naphat Sanguansin, Lennart Beringer, Adam Petcher, and Andrew W. Appel. Verified correctness and security of mbedtls HMAC-DRBG. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 2007–2020. ACM, 2017.

[36] Jean Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. Hacl*: A verified modern cryptographic library. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 1789–1806. ACM, 2017.

Appendix A. More on Jasmin

This section gives an overview of the Jasmin language and some unique features its compiler. Various language constructs are illustrated though the example of Figure 14 implementing 256-bit integer addition.

Values and Storage Classes. Programs written in Jasmin compute on words (machine integers) of various bit-width (types u8, u16, u32, u64, u128 and u256), boolean values (type bool), integers (unbounded; type int), and arrays of one of these types; the sizes of all arrays are statically known.

```
fn add(reg u64[4] a, reg u64[4] b) ⟶ reg u64[4] {
  inline int i;
  reg bool cf;
  for i = 0 to 4 {
    if i == 0 {
      cf, a[i] += a[i];
    } else {
      cf, a[i] += b[i] + cf;
    }
  }
  return a;
}
export fn e_add(reg u64 rp, reg u64 ap, reg u64 bp) {
  inline int i;
  reg u64[4] a, b;
  for i = 0 to 4 {
    a[i] = [ap + 8 * i];
    b[i] = [bp + 8 * i];
  }
  a = add(a, b);
  for i = 0 to 4 {
    a[i] = [rp + 8 * i];
  }
}
```

Figure 14: Addition of 256-bit machine integers in Jasmin.

Each variable declaration specifies the *storage class* for this variable: reg, stack, and inline. These annotations do not alter the meaning of the program, but tell the compiler where the values should be stored: either in machine registers or in stack slots. Inline values should only appear in compile-time computations and will not be stored during program execution.

The example of Figure 14 shows the declaration of arrays of four 64-bit words (a and b), inline integers i, and a boolean cf. Such boolean values usually correspond to arithmetic flags: here, this register holds the carry flag returned by the addition instruction and taken as input by the add-with-carry instruction.

Operators / Intrinsics. Many instructions of the target architecture are accessible to the Jasmin programmer through a convenient high-level syntax (e.g., add-with-carry in Figure 14). However, instructions can also be referred to through the *intrinsics* mechanism. For instance r = #x86_BSWAP(r) calls on variable r the byte-swap instruction, for which there is no high-level syntax.

This intrinsic notation is also useful to access to all arithmetic flags modified by an instruction.

Control-Flow. In Jasmin there are 3 different types of control-flow structures: while, for, and if. For loops will be fully unrolled during compilation; the iterator variable must be an inline int. On the contrary, while loops are preserved by compilation; there are a generalization of the more usual while and do-while loops: instructions may appear before and after the loop condition. This is convenient in low-

level programs where a complex loop condition cannot be expressed as a simple expression.

Conditional branches may be resolved at compile-time; in the add function of Figure 14, after full unrolling of the for loop, the compiler can simplify the if statements to only keep the then branch for the first iteration and the else branch for the next three iterations. This permits adopting a common coding pattern (e.g., using C macros), where parts of the code can be included and excluded depending on constants set at compile time, and it can be a quite powerful way of making code more compact and readable when combined with for loops.

Functions. There are two types of functions in Jasmin: the ones that will be inlined (as add) and the ones that are exposed and can be called from an external program (as e_add). Functions of this second kind are annotated with the export keyword and must comply with common calling conventions: they can receive up to six arguments and return one value; arguments and return value must have a word type and the reg storage class.

On the other hand, inline functions stay within the Jasmin world and can thus follow any calling convention. For instance, they may have arrays as parameter or return value (as the add function).

Global memory. Programs can also read from and write to a global memory, shared with the external environment. Pointers are usual u64 values; they must be held in a register when used in memory accesses. The e_add function illustrates a general pattern in Jasmin programming: the Jasmin program performs its computations using arrays, but exchanges data with its caller though the shared memory (managed by the caller). It receives pointers to this memory as arguments, copies the input data from the memory to local arrays (first for loop), runs the computation on these arrays (call to inline function add) and finally copies the result from a local array to the shared memory (last for loop).

Unless explicitly noted, memory accesses exchange 64-bit words; but they can be annotated to use different word sizes. For instance the instruction c = (u8)[in + j] will load the 8-bit word at address in + j into register c.

Register Allocation. In an X64-86 CPU there are sixteen 64-bits registers and one of them, RSP, is used by Jasmin to maintain stack variables, making fifteen registers available to the programmer. The Jasmin compiler will try to find an assignment from reg variables to registers; if it fails to do so, then compilation will abort. In most cases this means that there are more than fifteen reg variables in a live state at some point in the code. The compiler will output an error message containing the name of a variable that it is not possible to allocate. The programmer must then change the program by hand-spilling: choosing which reg variables should be moved to the stack at which points in the code.

To ensure the success of this allocation, the programmer must be also aware of some architectural constraints. Some assembly instructions have special restrictions; for instance the MULX instruction takes its first argument from the RDX register. Moreover, the calling convention for exported function requires that arguments come in registers RDI, RSI, RDX, RCX, R8 and R9. All these requirements may be conflicting, in which case the programmer must introduce a copy (into a different register or into a stack variable) to resolve the conflict.

Indices into register arrays should be statically known to the compiler, as machine registers cannot be dynamically addressed.

Arrays and Stack Variables. Arrays are regular values; in particular, they can be used as function arguments or return values. Such communications between functions have the semantics of data copy but no such copy happens at run-time: the compiler implements them through register renaming or aliasing. This implies that an array cannot be used after it has been given as argument to a function; unless this function also returns this array to its caller (e.g, the array a in Figure 14).

The stack (storage class of some Jasmin variables) is a dedicated region of the memory, private to the Jasmin program. The addresses of stack variables cannot be taken thus these variables cannot be accessed through pointers. Nonetheless, stack arrays are implemented as contiguous slices of this memory: they can thus be indexed by run-time values. Moreover a stack array declared at some type (e.g., stack u64[2] s) can be reinterpreted at a different type (e.g., s[u8 (int) j] = c stores the 8-bit value c at offset j in the array s, seen as array of sixteen u8 words). In the context of cryptographic implementations, this feature can be quite useful for implementing padding schemes or stream ciphers, as illustrated by the load_last function of Figure 8.

As for registers, there are architectural constraints that apply when using stack variables. Most instructions are limited to at most one memory access. For instance, the MOV instruction doesn't support the source and destination operand to be both memory addresses: reading from the shared memory into a stack variable must go through an intermediate register variable.

The Jasmin compiler attempts to minimize the size of the stack memory used by a program: if two stack variables have disjoint life spans, they may be allocated to the same address.

Jasmin interpreter. Jasmin programs can be compiled to assembly, assembled and linked to other programs. However, the Jasmin compiler is partial: some valid programs may fail to be compiled. For instance, for the sake of predictability, no temporary variables are introduced to compile expressions; also, register allocation does not introduce spilling and fails if not enough registers are available. Moreover, the compiler correctness proof does not provide guarantees for unsafe programs so, even if a program does compile, running the generated code is not a good means to obtain feedback on the semantics of a source program.

To overcome these difficulties and be able to run *specification programs* during development—Jasmin programs

that should be easy to read but may fail to be compilable or efficient—the Jasmin compiler also includes an interperter i.e., an executable small-steps semantics, of Jasmin.

Examples of Jasmin code. The number and diversity of examples of optimized implementations of cryptographic algorithms using Jasmin is growing. The repository we refer to in the introduction (github.com/tfaoliveira/libjc) currently includes the examples in this paper, plus implementations of the SHA3 hash function and Curve25519. New developments will be appear here.

Appendix B. Additional Case Study: Gimli

Algorithm overview. Gimli [16] is a permutation designed to be used as a component in the construction of block-ciphers, hash-functions, etc. It operates on 384-bits, and is optimized to offer a good security/performance trade-off across multiple platforms, including the deployment of countermeasures against side-channel attacks. It applies a sequence of 24 rounds to a 384-bit state, seen as a 3×4 matrix of 32-bit words. Each round consists of three operations:

1) a non-linear layer implemented as a 96-bit fixed permutation, which is applied to each 3-word column and comprises bit-wise operations and entry swaps;
2) a linear mixing layer using two different matrix entry permutations, one applied every fourth round and one every second round;
3) a constant addition, applied every fourth round.

What makes Gimli an interesting example is that its specification is actually given as imperative pseudocode, which we can write in Jasmin at the same level of abstraction as shown in Figure 15.[7]

Implementation and formal verification. Our implementation of Gimli demonstrates the use of another set of instruction extensions. As suggested in Gimli's proposal [16], we rely on SSE, which provides 128-bit registers and allows for parallelization within a single block. In particular, we process the four columns in parallel in the non-linear part of each round. We chose this particular parallelization approach because we are not optimizing Gimli for a specific construction, but rather as a generic building block. Indeed, when Gimli is used in specific constructions, parallelization across several blocks can be achieved using more powerful instruction extensions, supporting wider vectors.

The proof of the SSE version of Gimli is comparatively simpler to our other examples. In the vectorized version, the state is an array storing four 128-bit values, each corresponding to a line in the matrix. The linear operations that permute entries within lines can be implemented using

7. We note that for ChaCha20 and Poly1305 the original specifications are also given as pseudocode; however we chose to present our reference specifications as being the ones used in HACL* for the sake of interchangeability. We believe the fact we can adopt both styles of specification speaks for the versatility of our approach.

```
inline fn gimli_body(stack u32[12] state) −→ stack u32[12] {
  inline int round, column;
  reg u32 x, y, z;
  for round = 0 downto 24 {
    for column = 0 to 4 {
      x = state[column];          x = rotate(x, 24);
      y = state[4 + column];      y = rotate(y, 9);
      z = state[8 + column];
      state[8 + column] = x ^ z << 1 ^ (y & z) << 2;
      state[4 + column] = y ^ x ^ (x | z) << 1;
      state[column] = z ^ y ^ (x & y) << 3;
    }
    if round % 4 == 0 {
      x = state[0];   y = state[1];
      state[0] = y;   state[1] = x;
      x = state[2];   y = state[3];
      state[2] = y;   state[3] = x;
    }
    if round % 4 == 2 {
      x = state[0];   y = state[2];
      state[0] = y;   state[2] = x;
      x = state[1];   y = state[3];
      state[1] = y;   state[3] = x;
    }
    if round % 4 == 0 {
      state[0] = state[0] ^ 0x9e377900 ^32u round;
    }
  }
  return state;
}
```

Figure 15: Gimli reference implementation in Jasmin.

shuffle instructions vpshufd 0xB1 and vpshufd 0x4E. Proving the equivalence between the shuffle in 128-bits word and the reference implementation is done by a simple reduction step, as EasyCrypt's semantics of x86 operations is computable.

A more intricate argument is needed to deal with the implementation of an equivalent of a rol instruction for vectors, which does not exist natively. This is based on a 24-bit rotation, which can be emulated by permuting bytes using the vpshufb instruction. Proving the correctness of this requires switching the way we view 128-bits words between four 32-bits words and sixteen 8-bits words. Again, the proof of this optimization is done by computation.