Jasmin: High-Assurance and High-Speed Cryptography

José Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, Pierre-Yves Strub

HAL Id: hal-01649140

https://hal.archives-ouvertes.fr/hal-01649140

Submitted on 27 Nov 2017

Jasmin: High-Assurance and High-Speed Cryptography

José Bacelar Almeida
INESC TEC and
Universidade do Minho, Portugal

Manuel Barbosa
INESC TEC and FCUP
Universidade do Porto, Portugal

Gilles Barthe
IMDEA Software Institute, Spain

Arthur Blot
ENS Lyon, France

Benjamin Grégoire
Inria Sophia-Antipolis, France

Vincent Laporte
IMDEA Software Institute, Spain

Tiago Oliveira
INESC TEC and FCUP
Universidade do Porto, Portugal

Hugo Pacheco
INESC TEC and
Universidade do Minho, Portugal

Benedikt Schmidt
Google Inc.

Pierre-Yves Strub
École Polytechnique, France

ABSTRACT

Jasmin is a framework for developing high-speed and high-assurance cryptographic software. The framework is structured around the Jasmin programming language and its compiler. The language is designed for enhancing portability of programs and for simplifying verification tasks. The compiler is designed to achieve predictability and efficiency of the output code (currently limited to x64 platforms), and is formally verified in the Coq proof assistant. Using the SUPER-COP framework, we evaluate the Jasmin compiler on representative cryptographic routines and conclude that the code generated by the compiler is as efficient as fast, hand-crafted, implementations. Moreover, the framework includes highly automated tools for proving memory safety and constant-time security (for protecting against cache-based timing attacks). We also demonstrate the effectiveness of the verification tools on a large set of cryptographic routines.

CCS CONCEPTS

• **Security and privacy** → **Software security engineering**; **Logic and verification**; **Software security engineering**;

KEYWORDS

cryptographic implementations, verified compiler, safety, constant-time security

1 INTRODUCTION

Cryptographic software is pervasive in software systems. Although it represents a relatively small part of their code base, cryptographic software is often their most critical part, since it forms the backbone of their security mechanisms. Unfortunately, developing high-assurance cryptographic software is an extremely difficult task. Indeed, good cryptographic software must satisfy multiple properties, including efficiency, protection against side-channel attacks, and functional correctness, each of which is challenging to achieve:

• *Efficiency.* Cryptographic software must imply *minimal overhead* for system performance, both in terms of computational and bandwidth/storage costs. These are first-class efficiency requirements during development: a few clock-cycles in a small cryptographic routine may have a huge impact when executed repeatedly per connection established by a modern service provider;

• *Protection against side-channel attacks.* In-depth knowledge of real-world attack models, including side-channel attacks, is fundamental to ensure that the implementation includes adequate mitigation. For example, one must ensure that the observable timing behavior of the compiled program does not leak sensitive information to an attacker. Failing to address these considerations is a major attack vector against cryptographic implementations [1, 12]. Indeed, one prevailing view is that critical code must adhere to the "cryptographic constant-time" discipline, in particular its control flow and sequence of memory accesses should not depend on secrets [12]. High-assurance cryptographic software must be guaranteed to correctly adhere to this discipline.

• *Functional correctness.* Specifications of cryptographic components are often expressed using advanced mathematical concepts, and being able to bridge the enormous semantic gap to an efficient implementation is a pre-requisite for the implementor of a cryptographic component. Moreover, implementations may involve unconventional tasks, such as domain-specific error handling techniques. Guaranteeing functional correctness in these circumstances is harder than for other software domains, but it is critical that it is guaranteed from day 1—contrarily to the usual detect-and-patch approach—as implementation bugs in cryptographic components can lead to attacks [18, 23].

Efficiency considerations rule out using high-level languages, since the code must be optimized to an extent that goes far beyond what is achievable by modern, highly optimizing, compilers. Furthermore, there are concerns that highly optimizing compilers may introduce security flaws [21, 26]. As a consequence, the development of cryptographic software must be carried at assembly level, and is entrusted to a few select programmers. Moreover, these programmers rely on rudimentary tooling, that is often co-developed with the implementations themselves. For instance, security- and performance-critical parts of the OpenSSL library result from an *ad hoc* combination of pseudo-assembly programming and scripting, known as "perlasm". Another alternative is to use the qhasm language [11],
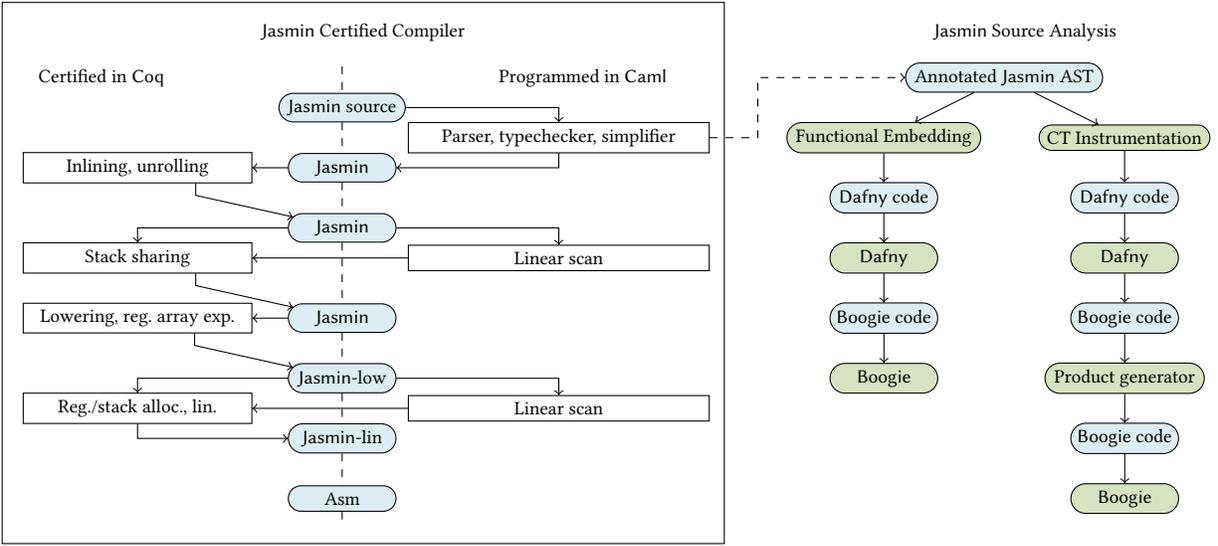
Figure 1: Jasmin architecture.

that simultaneously elides low-level details that are inessential for fine-grained performance tuning, and retains all performance- and security-critical aspects of assembly programming. qhasm achieves an excellent balance between programmability and efficiency, as evidenced by a long series of speed-record-breaking cryptographic implementations. Due to their nature, these approaches do not lend themselves to being supported by formal verification.

Functional correctness and side-channel security requirements for high-assurance cryptography impose going significantly beyond the current practices used for validating implementations, namely code inspection, code testing (and in particular, fuzzing), and even static analysis. Code inspection is time-consuming and requires a high-level of expertise. Testing is particularly effective for excluding errors which manifest themselves frequently, but performs poorly at detecting bugs which occur with very low probability. Static analysis is useful for detecting programming errors, but does not discover functionality bugs. A better alternative is to create machine-assisted verification frameworks that can be used for building rigorous proofs of functional correctness and side-channel security. However, and as stated above, these frameworks are not easily applicable to assembly languages.

Our contribution. We propose a tool-assisted framework, called Jasmin, for high-speed and high-assurance cryptographic code. Jasmin is inspired by qhasm, but it specifically addresses the lack of independent validation that exists today and enables the creation of *high-assurance* high-speed and high-security software that can be used as a drop-in replacement for unverified routines in existing cryptographic libraries. Specifically, the Jasmin framework goes significantly beyond current practices in cryptographic engineering, by leveraging state-of-the-art methods from programming languages, without sacrificing efficiency considerations.

More technically, we make the following contributions:

- we define the Jasmin programming language. Jasmin is designed to significantly simplify the writing and verification of high-speed cryptographic programs. In particular, Jasmin supports

within one single language: high-level features, including structured control flow such as loops and procedure calls, which lead to compact code that is also easier to verify; and assembly-level instructions (both generic and platform-specific), which give programmers tight control over the generated code. We give a formal, machine-checked, semantics of Jasmin in the Coq proof assistant;

- we define and implement a formally verified compiler that transforms Jasmin programs into assembly programs. The compiler alternates between certified passes (function inlining, loop unrolling, constant propagation), which are proved and verified in Coq, and passes by translation validation (register allocation), which are programmed in a conventional programming language and whose results are checked in Coq. The compiler is carefully designed to achieve predictability, and to deliver efficient code;

- we define and implement a sound embedding of Jasmin programs into Dafny [27], and use the embedding to support automated proofs of memory safety, constant-time security, and (potentially) functional correctness of Jasmin programs. The tool uses Boogie [7] to generate verification conditions and Z3 [20] to discharge them; for constant-time security, we use product programs as in [4]; we have also a proof-of-concept direct translation to SMT-Lib which we have used to replicate the correctness proof strategy of [19] using Boolector [30].

- we validate our framework with an implementation of scalar multiplication for Curve25519, the core cryptographic component in key exchange and digital signature algorithms recently adopted for widespread use in both TLS and the Signal protocol. We prove that the Jasmin implementation is memory-safe and is constant-time. This case study also serves as a point of comparison with prior work [19], which pursues the same goal using general-purpose verification tools; this comparison highlights the advantages of having a single integrated formal verification framework for high-speed cryptography.

- we carry a practical evaluation of our tools on a representative set of examples, which comprises many qhasm implementations included in the SUPERCOP framework. To this end, we created a simple automatic translator from qhasm to Jasmin, which shows that one can actually use Jasmin to program in a style very similar to that used in qhasm. We benchmark the efficiency of the code generated by the Jasmin compiler and show that its efficiency matches the original implementations generated from qhasm.

Figure 1 provides a high-level view of the workflow of our toolchain. On the left-hand side one can see the internal structure of the Jasmin compiler, which takes Jasmin code and produces assembly code that can then be further compiled and linked to larger programs/libraries. The various internal passes of the compiler will be explained in Section 5. On the right-hand side one can see the tool-chain for reasoning about Jasmin at the source level. This comprises a tool that can perform two types of translations of Jasmin programs into Dafny [27], which will be described in detail in Section 4. The first translation, which we call *functional embedding*, translates Jasmin programs into Dafny programs with consistent axiomatic semantics, including safety assertions that will cause any unsafe Jasmin program to be rejected. This embedding also permits translating typical functional correctness annotations into the Dafny program and take advantage of the Dafny/Boogie verification condition generator to discharge the associated proof goals using the Z3 SMT solver;[1] The second translation, which we call *CT instrumentation*, creates a Dafny program that will be translated into a Boogie program with special annotations. These will subsequently be intercepted by a sister program (*product generator* in the figure), which produces a product program whose safety implies the constant-time security of the original Jasmin program, using essentially the same theoretical principles of ct-verif [4]. The Jasmin compiler is proven in Coq to preserve safety and correctness properties, and we sketch a manual proof in Section 5 that it also preserves the constant-time property.

Trusted Computing Base. The Trusted Computing Base (TCB) of the Jasmin framework currently includes Coq, the unverified parts of the Jasmin compiler (limited to parsing, type-checking and code pretty-printing), and the translator from Jasmin to Dafny code. Because we currently rely on Dafny for source-level verification, we also rely on the TCB of the Dafny verification infrastructure. Using a verified verification condition generator, together with foundational tools for constant-time and memory safety, would eliminate Dafny from the TCB.

Limitations. The emphasis of this work is on providing an end-to-end infrastructure for high-assurance and high-speed cryptography, and to demonstrate the effectiveness of automated methods for memory safety and constant-time security. We also provide support for proving functional correctness, but do not exercise this component of the framework over substantial examples, such as scalar multiplication of Curve25519, for two main reasons. First, verifying functional correctness with our current infrastructure would replicate prior work—which we discuss in Section 2—and in particular would involve a cumbersome, hand-managed, process of

combining SMT-based and interactive verification in the Coq proof assistant. Second, we are developing a verified verification condition generator, in the spirit of the Verified Software ToolChain [6], which provides an integrated and foundational environment for managing such proofs, and eventually connecting with existing mathematical formalizations of elliptic curves [9].

Moreover, Jasmin currently lacks features that are widely used in cryptographic implementations, e.g. floating-point arithmetic or vectorized instructions. Adding these instructions is orthogonal to the main contributions of this paper and is left for future work. Similarly, Jasmin currently supports a single micro-architecture, in contrast to qhasm and "perlasm" which support multiple ones. Nevertheless, we leave for further work to support different micro-architectures, and to provide stronger evidence that Jasmin offers (at least) the same level of portability.

Access to the development. The Jasmin framework can be obtained from https://github.com/jasmin-lang/jasmin.

2 MOTIVATING EXAMPLE

We will illustrate the design choices of the Jasmin framework and its workflow using a classic example from elliptic curve cryptography which we briefly introduce below.

A primer on elliptic curve cryptography. Elliptic curve cryptography [24] relies on hardness assumptions on algebraic groups formed by the points of carefully chosen elliptic curves over finite fields. Let $\mathbb{F}_q$ be the finite field of prime order q. An elliptic curve is defined by the set of points $(x, y) \in \mathbb{F}_q \times \mathbb{F}_q$ that satisfy an equation of the form $E : y^2 + a_1 xy + a_3 y = x^3 + a_2 x^2 + a_4 x + a_6$, for a_1, a_2, a_3, a_4, $a_6 \in \mathbb{F}_q$ (with certain restrictions on these parameters). This set of points, together with a "point at infinity", form a group of size $l \approx q$. The group law has a geometric interpretation, which is not relevant for the purpose of this paper; what *is* important is that the group law can be computed very efficiently—particularly when compared to the computations underlying other algebraic structures used in public-key cryptography—using only a few operations in $\mathbb{F}_q$. Similarly, scalar multiplication,[2] which is the core operation for elliptic curve cryptography, can also be computed very efficiently.

Curve25519. X25519 is an elliptic-curve Diffie-Hellman key exchange protocol proposed by Bernstein [13]. It is based on the custom-designed curve Curve25519 defined as $E : y^2 = x^3 + 486662x^2 + x$ over the field $\mathbb{F}_{2^{255}-19}$. This curve was chosen to provide cryptographic security, but design choices also took into consideration the need for aggressive optimization. As a result of these choices, Curve25519 has been adopted for widespread use in various contexts, including the TLS and the Signal protocols.

Scalar multiplication in Curve25519 is usually implemented using Montgomery's differential-addition chain—a.k.a. Montgomery ladder—which permits performing the computation directly over the x-coordinate of elliptic curve points. This algorithm is shown in Algorithm 1. It is ideal for high-security and high-speed implementation for two reasons. First, it is much simpler than the

---

[1] we have also developed a proof-of-concept translator to SMT-Lib in order to experiment with other SMT solvers, namely Boolector.

[2] Given a curve point P and a scalar $k \in \mathbb{Z}$, scalar multiplication computes the point $Q = k \cdot P = \underbrace{P + \ldots + P}_{k \text{ times}}$.

generic algorithm for elliptic curves, so its overall efficiency essentially only depends on the cost of the underlying field operations, which can be computed very fast in modern architectures. Second, it is highly regular and can be implemented in constant-time by executing exactly the same code for each scalar bit (called a *ladder step*), making sure that the appropriate inputs are fed to this code via (constant-time) swapping of (X_2, Z_2) with (X_3, Z_3). The computations of each step in the ladder, all over $\mathbb{F}_q$, are shown in Algorithm 2. Typical implementations of the scalar multiplication operation implement the Montgomery ladder step in fully inlined hand-optimized assembly, and also include field multiplication and inversion as hand-optimized assembly routines (these are needed to recover the final x-coordinate of the result once the ladder is computed). The main difference between various implementations lies in the representation of $\mathbb{F}_{2^{255}-19}$ field elements and their handling in the hand-crafted assembly code, as the optimal choice varies from one architecture to another due to word size and available machine operations, and their relative efficiency. The higher-level functions that call the assembly routines for the various ladder steps and finalize the results are usually implemented in C. This is inconvenient when formal verification is the goal, since the relevant routines are now split between two programming languages with very different characteristics.

Algorithm 1 Curve25519 Montgomery Ladder

Input: A scalar k and the x-coordinate x_P of a point P on E.
Output: (X_{kP}, Z_{kP}) fulfilling $x_{kP} = X_{kP}/Z_{kP}$
$t \leftarrow \lceil \log_2 k + 1 \rceil$
$X_1 \leftarrow xP; X_2 \leftarrow 1; Z_2 \leftarrow 0; X_3 \leftarrow x_P; Z_3 \leftarrow 1$
for $i \leftarrow t - 1$ **downto** 0 **do**
   if bit i of k is 1 **then**
      $(X_3, Z_3, X_2, Z_2) \leftarrow$ ladderstep$(X_1, X_3, Z_3, X_2, Z_2)$
   else
      $(X_2, Z_2, X_3, Z_3) \leftarrow$ ladderstep$(X_1, X_2, Z_2, X_3, Z_3)$
   end if
end for
return (X_2, Z_2)

---

3 JASMIN LANGUAGE

The claim of this paper is that it is possible to obtain the best of the two worlds, and to develop effective verification methodologies whose guarantees carry to assembly-level implementations. The key to achieving this goal is the Jasmin programming language, which is specifically designed to ease the writing and verification of high-speed code, and the Jasmin verified compiler, which ensures that properties of programs provably carry to their assembly implementations.

In this section, we detail the design rationale of the Jasmin language, and then give a formal overview of its syntax and semantics.

3.1 Language design

Figures 3 and 2 show two illustrative snippets of a Jasmin implementation of scalar multiplication for Curve25519. This example highlights one fundamental design goal of Jasmin: one can implement complete cryptographic primitives within a single language

---

Algorithm 2 One step of the Curve25519 Montgomery Ladder

---

function ladderstep$(X_1, X_2, Z_2, X_3, Z_3)$
   $T_1 \leftarrow X_2 + Z_2$
   $T_2 \leftarrow X_2 - Z_2$
   $T_7 \leftarrow T_2^2$
   $T_6 \leftarrow T_1^2$
   $T_5 \leftarrow T_6 - T_7$
   $T_3 \leftarrow X_3 + Z_3$
   $T_4 \leftarrow X_3 - Z_3$
   $T_9 \leftarrow T_3 \cdot T_2$
   $T_8 \leftarrow T_4 \cdot T_1$
   $X_3 \leftarrow T_8 + T_9$
   $Z_3 \leftarrow T_8 - T_9$
   $X_3 \leftarrow X_3^2$
   $Z_3 \leftarrow Z_3^2$
   $Z_3 \leftarrow Z_3 \cdot X_1$
   $X_2 \leftarrow T_6 \cdot T_7$
   $Z_2 \leftarrow 121666 \cdot T_5$
   $Z_2 \leftarrow Z_2 + T_7$
   $Z_2 \leftarrow Z_2 \cdot T_5$
   return (X_2, Z_2, X_3, Z_3)
end function

---

and use different programming idioms for different parts of the implementation. On the one hand, the ladder step is implemented as hand-optimized code, using a convenient and uniform syntax for instructions. This style of programming is close to qhasm, with each statement corresponding to a single processor instruction. On the other hand, the ladder itself uses high-level control-flow structures, including for and while loops, function calls, array notation and the passing of arrays as parameters. This style of programming leads to compact and intuitive code, and also greatly facilitates safety, side-channel and functional correctness verification. We detail these choices next.

Predictable pre-assembly programming. 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.

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. However, platform-specific instructions are also available and can be used whenever important, e.g., for efficiency. In particular, and similarly to qhasm, programmers may always use a Jasmin dialect where there is a strict one-to-one mapping between Jasmin instructions and assembly instructions. This is visible in Figure 2, where we show qhasm and corresponding Jasmin code side by side; these are snippets of the implementation of the ladder-step algorithm in Algorithm 2.

Finally, to ensure predictability, the programmer must also specify the storage for program variables (stack, register) and must handle spilling explicitly. However, full register naming is not needed; the programmer only needs to ensure that there *exists* a mapping

Figure 2: Snippets of Jasmin ladder step function (left) generated from qhasm (right).

```
export fn ladderstep(reg b64 workp) {     input workp
  reg b64 addt0;                          int64 addt0
  reg b64 addt1;                          int64 addt1
  reg bool cf;                            int64 t10
  reg b64 t10;                            int64 t11
  reg b64 t11;                            int64 t12
  reg b64 t12;                            int64 t13
  reg b64 t13;                            int64 t20
  reg b64 t20;                            int64 t21
  reg b64 t21;                            int64 t22
  reg b64 t22;                            int64 t23
  reg b64 t23;                            ...
  …                                       enter ladderstep
  t10 = [workp + 4 * 8];                  t10 = *(uint64 *)(workp + 32)
  t11 = [workp + 5 * 8];                  t11 = *(uint64 *)(workp + 40)
  t12 = [workp + 6 * 8];                  t12 = *(uint64 *)(workp + 48)
  t13 = [workp + 7 * 8];                  t13 = *(uint64 *)(workp + 56)
  t20 = t10;                              t20 = t10
  t21 = t11;                              t21 = t11
  t22 = t12;                              t22 = t12
  t23 = t13;                              t23 = t13
  cf, t10 += [workp + 8 * 8];             carry? t10 += *(uint64 *)(workp + 64)
  cf, t11 += [workp + 9 * 8] + cf;        carry? t11 += *(uint64 *)(workp + 72) + carry
  cf, t12 += [workp + 10 * 8] + cf;       carry? t12 += *(uint64 *)(workp + 80) + carry
  cf, t13 += [workp + 11 * 8] + cf;       carry? t13 += *(uint64 *)(workp + 88) + carry
  addt0 = 0;                              addt0 = 0
  addt1 = 38;                             addt1 = 38
  addt1 = addt0 if ! cf;                  addt1 = addt0 if !carry
  cf, t10 += addt1;                       carry? t10 += addt1
  cf, t11 += addt0 + cf;                  carry? t11 += addt0 + carry
  cf, t12 += addt0 + cf;                  carry? t12 += addt0 + carry
  cf, t13 += addt0 + cf;                  carry? t13 += addt0 + carry
  addt0 = addt1 if cf;                    addt0 = addt1 if carry
```

from register variables to machine registers (without spilling), but the actual mapping is later found by the compiler. At the source level, stack variables and register variables are interpreted simply as variables; the storage modifier is only used as advice for register allocation. In particular, at this level the memory is assumed to be disjoint from stack storage. The compiler will later refine this model and conciliate the fact that stack data must reside in memory as well.

Verifiability. Formal verification of low-level code is extremely hard, because of complex side-effects (e.g. shift instructions have side-effects on flags), unstructured control-flow, and flat structure (i.e. code is often created by copy-and-paste followed by variable or register renaming). Jasmin includes several features that avoid these issues and enable a streamlined formal verification workflow.

Jasmin ensures that side-effects are explicit from the program code, by not treating flags specially in the input language; instead, flags are simply boolean variables. This treatment of flags is illustrated in Figure 2, where the carry flag cf is declared as a boolean variable. The programmer is expected to ensure that writing and reading of these variables is consistent with the underlying machine instruction semantics, which is checked by the compiler using an extended form of register allocation. Treating flags as boolean variables allows all operators to be pure, and so state modifications are all explicit, e.g., code of the form $(x_1, \ldots, x_n) := op(e_1, \ldots, e_k)$ only changes (function local) variables x_i. This approach makes

verification of functional correctness and even side-channel security significantly simpler, since it avoids the intricacies of dealing with the side-effects associated with flags.[3]

Unlike qhasm, Jasmin supports function calls. The use of function calls is shown in Figure 3, where two functions are used for computing a single ladder step and for performing a constant-time swap. Function calls naturally lead to a style of programming that favours modularity, and thus is more easily amenable to modular verification. Functions are always inlined and must explicitly return all changed values.[4] The stack allocation procedure ensures that inlining is "zero-cost", so that the extra benefits of modular code writing and analysis comes with no performance penalty.

Jasmin also supports high-level control-flow structures, instead of jumps supported by qhasm. The use of control-flow structures can be seen in Figure 3, where a while loop is used to iterate over the bit representation of the scalar in constant-time. The choice of using high-level structures over jumps usually has no impact on the efficiency of the generated code; indeed, the translation to assembly, which is achieved by unrolling or trivial mapping to label-goto constructions, is simple enough to retain predictability and our structures are sufficient to express the control-flow typically found in cryptographic implementations. In contrast, 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.

Jasmin also supports functional arrays for describing collections of registers and stack variables. Figure 3 shows how arrays can be used to refer to various registers/stack positions holding the data as $x[i]$ rather than hardwired variable names such as $x1$, $x2$, etc. 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, which permits replicating within the Jasmin language coding techniques that are typically implemented using macros in C. For example, one can write a for-loop ranging over the number of limbs for representing an element of $\mathbb{F}_q$, as in Figure 3 whereas a qhasm programmer would unroll such a loop by hand using error-prone "copy-and-paste" (or write the code in C).

3.2 Language specification

This paragraph outlines the syntax of the Jasmin language. A formal description using BNF notation is available as Appendix A.

Types. Jasmin features a simple type system, including types b_i and $b_i[n]$ for i-bit values and for n-dimensional arrays of i-bit values, with $i \in \{1, 8, 16, 32, 64\}$ and $n \in \mathbb{N}$, and *int* for unbounded integers. Unbounded integers are used only for compile-time expressions. The type b_1 is used to represent booleans. The choice of

---

[3]The exception are, of course, memory accesses, which are handled in the standard way by treating memory as a large chunk of shared state that can be accessed in an array-like form. However, memory access is often very simple in cryptographic implementations (particularly those dealing with algebraic operations) and so this has relatively low impact in the formal verification effort.
[4]This is expected to change in future versions of Jasmin.

Figure 3: Constant-time Montgomery Ladder in Jasmin

```
fn set_int(reg b64 v) ⟶ reg b64[4] {
    reg b64[4] res;
    inline int i;
    res[0] = v;
    for i = 1 to 3 { res[i] = 0; }
    return res;
}

fn mladder(stack b64[4] xr, reg b64 sp) ⟶ stack b64[4], stack b64[4] {
    reg bool cf;
    reg b64 tmp1, tmp2, bit, swap, i, j;
    stack b64 prevbit, is, js, s;
    stack b64[4] x1, x2, z2, x3, z3;
    inline int k;
    x1 = xr;
    x2 = set_int(1);
    z2 = set_int(0);
    z3 = set_int(1);
    x3 = xr;
    prevbit = 0;
    j = 62;
    i = 3;
    while (i >=s 0) {
        is = i; tmp1 = [sp + 8 * i]; s = tmp1;
        while (j >=s 0) {
            js = j; tmp2 = s;
            bit = tmp2 >> j;
            bit = bit & 1;
            swap = prevbit;
            swap ^= bit;
            prevbit = bit;
            x2, z2, x3, z3 = cswap(x2, z2, x3, z3, swap);
            x2, z2, x3, z3 = ladderstep(x1, x2, z2, x3, z3);
            j = js; j -= 1;
        }
        j = 63; i = is; i -= 1;
    }
    return x2, z2;
}
```

the type system ensures that Jasmin values match the machine interpretation: types only distinguish between sizes, whereas signed and unsigned interpretations are associated with the semantics of operators. This is visible in the condition of the while loop in Figure 3.

Storage types. Storage types are used by the compiler to fix how values are stored in memory. Storage types are: store in stack stack, store in register reg, resolve at compile-time inline. Integers are resolved at compile-time by default. Storage types are used in variable and function declarations; see Figure 3. For instance, the declaration reg bool cf introduces a boolean variable cf that will be stored in a register. The type of mladder indicates that the function will read and return from/to the stack two b_{64} arrays of size 4.

Expressions. Expressions are built from variables using constants, operators, accessors for arrays and memory, casts from integers to b_{64}, and conditionals. Operators include integer constants, arithmetic $(+, -, *, /)$, comparison $(<, >, \leq, \geq, =)$, and logical $(\land, \lor, \neg)$ operators and a rich set of generic and platform-specific assembly-level operations: conditional move, signed/unsigned addition, add with carry, exact/truncated multiplication, increment/decrement, shifts, etc. Operators have a type which reflects their effect on flags; e.g. the type of shift operators is $b_{64} \times b_{64} \to b_{64} \times b_1 \times b_1 \times b_1 \times b_1$,

where the five booleans in the return type correspond to the flags modified by the instruction.[5] Syntactic sugar is used to offer a simplified syntax for common cases such as when only the carry flag is used at the operator output. Memory accesses are of the form $[x + e]$, where x is variable of type b_{64} representing the pointer and e is an expression of type b_{64} representing the offset. Array accesses are of the form $x[e]$, where x is a variable and e is an expression of type b_{64}. A simple type system is used to ensure that expressions are well-typed.

Statements. Statements of the language are built from assignments, function calls, conditionals, for and (some mild generalization of) while loops, and the usual sequencing of statement ";". Assignments are of one of the following form: $d = e$, or $d_1, \ldots, d_k = op(e_1, \ldots, e_\ell)$, where $d, d_1, \ldots, d_k$ are destinations and $e, e_1, \ldots, e_\ell$ are expressions. A destination is either a variable x, an array destination $x[e]$, a memory destination $[x + e]$ or an underscore _ when the result can be discarded. Note that operator calls can have multiple destinations, according to their type. Function calls are also of the form $d_1, \ldots, d_k = f(e_1, \ldots, e_\ell)$, where the number and nature of arguments and destinations is determined by the type of the function. The for loops have a body, a range of the form $(e_1 \ldots e_2)$ and an iteration flag indicating if the loop ranges over $e_1, e_1 + 1, \ldots, e_2$ or $e_1, e_1 - 1, \ldots, e_2$. More interestingly, the syntax of the while loops is of the form while c_1 (e) c_2, where c_1 is a statement that is executed before testing the expression e, and c_2 is the loop body. This extended syntax is useful to capture different control-flow structures, including do-while and while-do loops. It also enables the use of arbitrary programs to express the loop guards. Statements must be well-typed; for instance, assigned expressions must match the type of their destination and loop guards must be boolean.

Programs. Programs p are mappings from function names f to function declarations $p(f)$, comprising a command $p(f)_c$ and two lists of variables $p(f)_{param}$ and $p(f)_{res}$ describing, from the callee point of view, the variables in which the arguments will be stored and from which the return values will be fetched. Programs must be well-typed, following a typing discipline similar to that of statements.

Each function declaration is annotated with either *export* or *inline* (the default). Only the first ones are compiled down to assembly. Calls to the second ones are meant to be fully inlined in the caller's body; in that sense, these functions are a zero-cost abstraction provided to the programmer to structure the code.

3.3 Semantics

The behavior of Jasmin programs is given by a big-step operational semantics relating initial and final states. The semantics defines a partial function: for every initial state, there is at most one final state, reflecting that Jasmin programs have a deterministic behavior. Determinism of Jasmin programs is essential for predictability, and considerably simplifies the proof of correctness of the compiler, as discussed in Section 5. The semantics is fully formalized in the Coq proof assistant and is used as the basis for justifying the correctness of the compiler and of the source-level analyses (formally using the

---

[5]We currently do not model some infrequently used flags such as *AF* since they are not used in our examples.

Coq proof assistant in the first case, and on paper in the second case). The semantics is defined in the context of a program, which is used to resolve function calls. To keep our notation readable, this ambient program is only shown when defining the semantics of function calls, which must resolve the called function.

Values. Booleans are interpreted by the set $\{0, 1, \bot\}$, where $\bot$ is used to model the behavior of some operations, e.g. shifts, on flags. Other types have the intended semantics. For instance, the type b_i is interpreted as $\{0, 1\}^i$, and the type $b_i[n]$ is interpreted as $\{0, 1\}^i[n]$, for $i \in \{8, 16, 32, 64\}$. The set of values is obtained by taking the union of the interpretation of types.

States. States are pairs (m, ρ) consisting of a global memory m, shared between all functions, which maps addresses to b_i values, and a local environment ρ, specific to each function, mapping variables to values.

The environment is a functional map which associates to a variable (with a given type and identifier) a value (or an error). We use the usual notations $\cdot[\cdot]$ for map access and $\cdot[\cdot := \cdot]$ for map update.

Memory. We use an axiomatic type for memories. This type is equipped with operations for reading and writing, which take a size i and an address of type b_{64}, and read or store a b_i value (possibly returning errors). We use $\cdot[\cdot]$ for reading and $\cdot[\cdot \leftarrow \cdot]$ for writing: $m[e]_i = v$ means that in memory m, reading a value of type b_i from address e successfully returns the value v; $m[e \leftarrow v]_i = m'$ means that in memory m, writing the value v with type b_i successfully results in the memory m'. In both cases, the optional subscript corresponds to the size of the value in memory (default is 64).

Which addresses can or cannot be used by a Jasmin program is expressed through a predicate, $\text{valid}_i(m, p)$, that states that in memory m, pointer p can be safely used for accesses of size i. This predicate is under-specified; it is supposed to capture various architectural and system-specific constraints such as alignment requirements. The memory thus enjoys the following laws, which express the usual properties of memories:

- valid addresses can be read:

$$(\exists v, m[p]_i = v) \iff \text{valid}_i(m, p);$$

- valid addresses can be written:

$$(\exists m', m[p \leftarrow v]_i = m') \iff \text{valid}_i(m, p);$$

- a written value can be read back:

$$m[p \leftarrow v]_i = m' \implies m'[p]_i = v;$$

- a memory write leaves the remainder of the memory unchanged:

$$m[p \leftarrow v]_i = m' \implies$$
$$\text{disjoint}(p, i, p', i') \implies m'[p']_{i'} = m[p']_{i'}$$

where disjoint expresses that two address ranges do not overlap.

In addition to reading and writing, the Jasmin memory features a restricted form of dynamic allocation to model the life span of local variables: fresh memory is allocated on function entry, and released on function exit. This is modeled by means of two operations: alloc-stack(m, n) allocates a region of size n in memory m and

free-stack(m, n) frees the top-most region in memory m, of size n. Such stack-allocated memory regions are handled through their base pointer. This stack is specified through three intermediate operators:

- top-stack(m) returns the base pointer of the top-most stack region of memory m;
- caller(m, p) returns, if any, the previous region, in the stack, of region starting at p in memory m;
- frame-size(m, p) returns the size of the memory region starting at p, or nothing if it is not the base pointer of such a region in memory m.

Writing to memory leaves these three properties unchanged; allocating and freeing update these properties to maintain the stack structure. The frame-size property enables us to implement allocation and freeing through addition and subtraction on a global stack pointer, without explicitly building a linked-list of frames.

Finally, since the compiler needs to emit *valid* memory accesses, the memory model features an operation is-align(n, i) that tells whether a pointer with offset n is correctly aligned for memory access of size b_i. The axiomatization of this operation mandates that, for stack memory regions, deciding whether a pointer is valid amounts to checking whether its relative offset (within this region) is aligned and in bounds.

Expressions. The semantics of expressions is defined in the usual way, and is parametrized by a state $s = (m, \rho)$ as described above. The evaluation of the expression e in the state s is noted $[\![e]\!](s)$, and is defined in the usual way.

Even though we denote the evaluation of expressions as a function, their semantics is partial: ill-typed expressions never evaluate; out-of-bounds array accesses are not defined, as well as invalid memory accesses.

Destinations. We will use the $s[\cdot := \cdot]$ notation for storing values in both the memory and environment, depending on the destination. Most of the rules are standard; the ones used for writing into memory and arrays are described in Figure 4.

Statements. Formally, the semantics is captured by judgments of the form $c, s \Downarrow s'$, stating that executing command c on initial state s terminates in final state s'. The rules, which are mostly standard, to the exception of the rules for functional arrays and procedure calls, are described in Figure 4. Sequences are defined in a standard way, with skip the empty command.

Loops use two blocks of instructions, allowing to handle both do-while and while-do constructions at the same time: first the first block is executed, then the condition is evaluated: if it's false we leave the loop, otherwise we execute the second block and then repeat from the first block.

The semantics of for loops use another judgement $\Downarrow_{for}$, where $c, s \Downarrow_{for}^{i \in \ell} s'$ describes the execution of the command c from the state s to the state s' with the i integer variable taking all the values in ℓ. If the list ℓ is the empty list $[]$, then the resulting state is the original state s. Otherwise, if the list ℓ has a head w and tail ws, the resulting state is the one after executing c in the state where the value w is assigned to the variable i, and then the rest of the values.

Figure 4: Semantics of the Jasmin language

$$(m, \rho)[[x + e] := v] = (m[\rho[x] + [\![e]\!](m, \rho) \leftarrow v], \rho)$$

$$(m, \rho)[x[e] := v] = (m, \rho[x := \rho[x][[\![e]\!](m, \rho) := v])$$

$$\overline{d = e, s \Downarrow s[d := [\![e]\!](s)]}$$

$$\frac{op([\![e_1]\!](s), \dots, [\![e_\ell]\!](s)) = (v'_1, \dots, v'_k)}{d_1, \dots, d_k = op(e_1, \dots, e_\ell), s \Downarrow s[d_j := v'_j]}$$

$$\overline{\text{skip}, s \Downarrow s} \qquad \frac{i, s \Downarrow s_1 \quad c, s_1 \Downarrow s'}{i; c, s \Downarrow s'}$$

$$\frac{c_1, s \Downarrow s' \quad [\![e]\!](s) = \text{true}}{\text{if } (e) \text{ then } c_1 \text{ else } c_2, s \Downarrow s'} \qquad \frac{c_2, s \Downarrow s' \quad [\![e]\!](s) = \text{false}}{\text{if } (e) \text{ then } c_1 \text{ else } c_2, s \Downarrow s'}$$

$$\frac{c_1, s_1 \Downarrow s_2 \quad [\![e]\!](s_2) = \text{true} \quad c_2, s_2 \Downarrow s_3 \quad \text{while } c_1 \ (e) \ c_2, s_3 \Downarrow s_4}{\text{while } c_1 \ (e) \ c_2, s_1 \Downarrow s_4}$$

$$\frac{c_1, s_1 \Downarrow s_2 \quad [\![e]\!](s_2) = \text{false}}{\text{while } c_1 \ (e) \ c_2, s_1 \Downarrow s_2} \qquad \frac{c, s \Downarrow_{for}^{i \in \text{range}([\![e_{lo}]\!](s), [\![e_{hi}]\!](s))} s'}{\text{for}(i = e_{lo} \text{ to } e_{hi})c, s \Downarrow s'}$$

$$\overline{c, s \Downarrow_{for}^{i \in []} s} \qquad \frac{s[i := w], c \Downarrow s_2 \quad c, s_2 \Downarrow_{for}^{i \in ws} s'}{c, s \Downarrow_{for}^{i \in w::ws} s'}$$

$$\frac{a, (m, \rho) \Downarrow v_a \quad f, v_a, m \Downarrow_{call}^{p} v_r, m'}{r = f(a), (m, \rho) \Downarrow (m', \rho[r := v_r])}$$

$$\frac{p(f)_c, (m, \emptyset[p(f)_{param} := v_a]) \Downarrow (m', \rho')}{f, v_a, m \Downarrow_{call}^{p} [\![p(f)_{res}]\!](m', \rho'), m'}$$

Also, the function calls use the judgement $\Downarrow_{call}$ which describes the behavior of a function from the callee point of view: $f, v_a, m \Downarrow_{call}^{p} v_r, m'$ means that the function named f of the program p executed from the memory m with arguments v_a returns values v_r in the memory m'. Note that as said earlier, the environment between the caller and the callee are completely independent.

3.4 Memory safety and constant-time security

Two essential properties of Jasmin programs are memory safety and constant-time security. In this paragraph, we introduce the two notions. Later, we will argue that both notions are preserved by compilation, and present automated methods for verifying that Jasmin programs are memory safe and constant-time.

Memory safety. The Jasmin memory model is parameterized by the notion of validity. It must capture the various architectural constraints (e.g., alignment of pointers), and requirements from the execution environment (some memory regions might be reserved to the operating system, or to load the code text, which cannot be overwritten by Jasmin programs). Also, the allocation routine that is used to reserve memory for local variables on function entry must return fresh, valid memory addresses (or fail).

A Jasmin command c is safe if its semantics is defined for every initial state. Formally, $\forall s \cdot \exists s' \cdot c, s \Downarrow s'$. This notion is rather strong; in particular, it entails that the program is well-typed, that it

terminates on all inputs, that array accesses are always in-bounds, and that all memory accesses target *valid* memory.

This definition of safety being very strong, we assume that programs may be equipped with preconditions that restrict the set of admissible initial states: enough free stack space, validity of input pointers, etc.

Constant-time security. The semantics of Jasmin program can be instrumented to produce a leakage trace that records the branches that are taken and memory read/write operations performed during execution. Judgments of the extended semantics are of the form

$$c, s \Downarrow s', \ell$$

where ℓ is the leakage trace. The extended semantics is used to formalize constant-time programs: specifically, a Jasmin command c is constant-time iff for every states s_1, s_2, s'_1, s'_2 and leakage traces ℓ_1 and ℓ_2, we have:

$$\left. \begin{array}{c} c, s_1 \Downarrow s'_1, \ell_1 \\ c, s_2 \Downarrow s'_2, \ell_2 \\ s_1 \sim s_2 \end{array} \right\} \Rightarrow \ell_1 = \ell_2$$

where $\sim$ is an equivalence relation between states—as usual, $\sim$ is defined from security annotations specifying where secrets are held in the initial memory.

We stress that our notion of constant-time security is termination-insensitive and does not impose any condition on the program safety. However, it is well-known that naive error management is a main source of attacks in cryptographic implementations. In practice, we always check that programs are both safe and constant-time.

4 SAFETY AND CONSTANT-TIME ANALYSES

This section describes how Jasmin source level analyses are deployed on top of the Dafny verification infrastructure. An annotated Jasmin intermediate program is translated into two complementary Dafny programs: the first encodes safety; the second, assuming safety, encodes constant-time security policies as annotations.

4.1 Safety analysis

Dafny is a general-purpose verification language with support for procedures, loops, arrays and native bitvector theories. Our *functional embedding* of Jasmin into Dafny thus preserves the original program structure and is almost one-to-one. Most noteworthy, Jasmin functional arrays are encoded as fixed-size Dafny sequences, and memory encoded as a global array of byte blocks; reads and writes to memory are segmented into byte-wise operations, and require memory regions to be valid, axiomatized in Dafny as two ghost annotations Valid and ValidRange. Jasmin expressions and instructions are defined using mathematical integer and bitvector arithmetic, as precluded by the Coq semantics. An annotated snippet and its translation are shown in Figure 5.

The safety of the Jasmin program is therefore reduced to the safety of the functional Dafny embedding. The Dafny verifier guarantees that a safe program terminates and is free of runtime errors such as memory accesses, array indices or shift amounts out of bounds or division by zero.

Figure 5: Annotated Montgomery Ladder in Jasmin (left) and Dafny translations for safety (middle) and constant-time (right).

```
...
j = 62; i = 3;
while (i >=s 0)
//@ decreases i;
//@ invariant i <=s 4;
//@ invariant j == 62 || j == 63;
//@ invariant i >=s 0 ==> valid(sp, 8 * i, 8 * i + 7);
//@ security invariant public(i);
{
  is = i; tmp1 = [sp + 8*i]; s = tmp1;
  while (j >=s 0)
  //@ decreases j;
  //@ invariant j <=s 63;
  {
    ...
    j=js; j -= 1;
  }
  j = 63; i = is; i -= 1;
}
...
```

```
...
j := 62; i := 3;
while (i >=s 0)
decreases i; invariant (4 - i) >> 63 == 0;
invariant (j == 62) || (j == 63);
free invariant sp == old(sp);
invariant i >> 63 == 0 ==>
    ValidRange(sp as int + 8 * i as int, sp as int + 8 * i as int + 7);
{
  is = i;
  assert Valid(sp as int + 8 * i as int + 0); ...;
  assert Valid(sp as int + 8 * i as int + 7); ...;
  while (j >> 63 == 0)
  decreases j; invariant (63 - j) >> 63 == 0;
  { ...
    j = js; j := j - 1;
  }
  j := 63; i = is; i := i - 1;
}
...
```

```
...
j := 62; i := 3;
while (i >=s 0)
free ...
invariant Public(sp);
invariant Public(i); invariant Public(i > 1);
{
  assert Public(sp as int + 8 * i as int + 0); ...;
  assert Public(sp as int + 8 * i as int + 7);
  ...
  while (j >> 63 == 0)
  free ...
  invariant Public(j); invariant Public(j >> 63 == 0);
  {
    ...
    j = js; j := j - 1;
  }
  j := 63; i = is; i := i - 1;
}
...
```

For simple straight-line Jasmin programs, including most of our benchmarks, with no procedure calls, all loops unrolled and no memory operations, safety analysis can be performed fully automatically. Nevertheless, for more modular procedures, less well-behaved while loops or memory operations, programmers can supply additional annotations describing procedure contracts, loop invariants and valid memory regions. They can also express intermediate functional correctness properties seamlessly in the Jasmin annotation language. Taking a glance at Figure 5, both loops need invariants stating that the indices i and j decrease until zero within the loop and that they stay within bounds. Moreover, the 64 bits addressed by sp + 8*i need to constitute a valid memory region.

Under the hood, the Dafny verifier checks for correctness by translating to Boogie intermediate code. The Boogie verifier then generates verification conditions that are passed to the Z3 SMT solver. Alternatively, we have implemented a specialized verification condition generator for straight-line Jasmin procedures, in the style of [19]. This is more effective (specifically yields smaller verification conditions) for proving certain correctness properties of Jasmin programs, and targets specific SMT solvers such as Boolector that excels for bitvector arithmetic. We rely on the Haskell SBV[6] library as a universal interface with SMT-Lib.

4.2 Constant-time analysis

For constant-time analysis we instrument the generated Dafny program with special Public annotations on control flow conditions, memory accesses and array accesses, entailing that they do not depend on secrets. (Jasmin conditional assignments are compiled to constant-time instructions, so they do not require such safeguards.) As for safety, programmers can express additional security properties in Jasmin as boolean assertions using the public predicate. In the example from Figure 5, safety invariants are assumed to hold (marked with free or assume in Dafny) and the [sp + 8*i] memory read requires two security invariants stating that the values of sp and i are public inside the outer loop; the former is inferred from the procedure contract, and the second must be explicitly supplied by the programmer.

The constant-time instrumentation departs from the functional embedding described in the previous paragraph in the sense that we explore the existing translation from Dafny to Boogie to propagate constant-time security policies from Jasmin to Boogie programs. The Boogie input language has a well-defined semantics and was designed to be a convenient backend for verification tools. There, procedures are defined as a sequence of basic blocks that start with a label, contain straight-line statements with no if or while statements, and may jump at the end. To verify constant-time, we adapt a technique used by the ct-verif tool [4] which reduces the constant-time security of a Boogie program to the safety of a product program that emulates two simultaneous executions of the original program. We implement a Boogie-to-Boogie transformation tailored to the translation of Jasmin programs that computes the *product* of each procedure by essentially making shadow copies of program variables and duplicating all statements inside basic blocks to mention shadow variables instead, with two exceptions:

(1) procedure call statements are converted to single statements calling the product procedure with twice as many inputs and outputs, and

(2) assertions corresponding to constant-time security policy annotations are translated to relational assertions expressing first-order logic formulas that relate original and shadowed variables, by translating public(e) expressions to equality expressions e == e.shadow.

Proving that the product program is safe ensures that the equalities are always valid and suffices to conclude that the original program is sound. The soundness of this technique, that has been formally proven in Coq in [4], hinges on the assumption that program paths are independent from program secrets, guaranteeing that the product program always has as many paths as the original program and can therefore be efficiently verified.

5 CERTIFIED COMPILER

This section first describes the architecture of the Jasmin compiler and the various compilation passes. It then states and explains its main correctness theorem and describes the proof methodology.

Finally we argue that the compilation preserves the constant-time property of the programs it processes.

5.1 Compilation passes

The Jasmin compiler is formally verified. Therefore, it is mainly written using the Coq programming language. However, some parts are written in OCaml. The diagram shown on the left of Figure 1 summarizes its internal architecture.

The first passes parse the source program and enforce some typing rules. Then, the *constant expansion* pass replaces parameters by their values. Parameters are named constants whose values are known at compile-time.

Inlining replaces the function calls that are labeled inline with the body of the called function. In function calls, functions are designated by their name, thus statically determining which function is called (i.e., there are no function pointers). Also, only functions that are defined before a call-site can be inlined at this site. Therefore, (mutually) recursive functions cannot be fully inlined and recursive inlining always terminates. After inlining, the definitions of functions that are neither called nor exported are removed.

The inlining pass also introduces (copy) assignments to model communication (arguments and returned values) between the caller and the inlined callee. Thus, a renaming pass, akin to register allocation, tries to eliminates these assignments. This is particularly relevant when arguments and returned values include arrays: no copy should be introduced by the compiler. This means that the use of arrays must be linear: when an array is given as argument to a function, it can no longer be used by the caller, unless the callee returns it. As at most one copy of each array is live at the same time, no copy is ever needed.

The *unrolling* pass fully unrolls for-loops, whose bounds must always be statically determined. Notice that in case of nested loops, the actual value of some of these bounds may only be known after constant propagation. Therefore, this pass iterates a sequence of unrolling, *constant propagation*, and *dead-code elimination* until a fixed point is reached (or some maximal number of iterations is reached). Note that unrolling may insert new assignments in the program to set the value of the loop counter on each iteration. These assignments are labeled with a special internal inline tag. The next pass, *inline assignment allocation*, eliminates these assignments by a form of register allocation.

The next pass performs *sharing of stack variables*. This pass optimizes the memory layout of local variables: variables that are never alive at the same time can be allocated to overlapping stack regions. This is a form of register allocation. The *constant propagation* optimization is run again after this pass.

The *register array expansion* pass translates arrays (which in Jasmin are meant to represent a collection of registers or contiguous memory addresses) into register variables or stack variables. This requires that all accesses to the arrays are done through statically determined indices.

The *lowering* pass translates high-level Jasmin instructions into low-level, architecture-dependent instructions. Because Jasmin features both low-level and high-level instructions, lowering does not require to switch between intermediate representations; however, it is expected that all high-level instructions are removed after this pass. This pass is also responsible for translating conditional expressions into the flag-based conditional instructions that are featured by the target architecture. This is examplified by the case of the AddCarry instruction: it takes three arguments (two machine integers and a carry) and returns two values (one machine integer and a carry). It is a high-level instruction, and therefore represents all additions of machine integers, with or without carry. The x64 architecture instruction set features, in particular, the operators ADD and ADC to perform additions. Only the second form receives an explicit carry as argument. Both return one machine integer and affect many boolean flags. The lowering pass thus replaces the generic AddCarry with the adequate ADD or ADC machine instruction, depending on the expression that is given as the carry argument.

The *register allocation* pass renames variables to match the names of the architecture registers. This pass does not attempt to spill any variable to memory; it checks that there exists a mapping from program variables that are labeled with the register storage modifier to architecture registers, and infers such a mapping. The compiler fails if no such mapping exists. Register allocation takes into account the various architectural constraints: some instructions require the output register to be the same as one of the arguments; some instructions require that some operands are assigned to specific registers. This last constraint includes the correct handling of flags.

The register allocation is performed by a simple, greedy algorithm, derived from *linear scan* [32]. The resulting allocation is flow-insensitive: within a given function, a variable is always allocated to the same register. To compensate for this limitation, a dedicated pass renames variables according to their liveness ranges, using an SSA-like form. Once again, the *constant propagation* pass is applied to eliminate instructions that have no effect.

The *stack allocation* pass puts all local (stack) variables into a single memory region that is allocated on function entry and freed on function exit. It also takes care of the remaining arrays: each access to a stack array is thus translated as a corresponding memory operation. At the end of this pass, all arrays have been removed from the program.

The previous pass marks the end of the middle-end transforming the structured intermediate representation of Jasmin. The *linearization* pass, transforms the program into an unstructured list of instructions with named labels and gotos. This representation is then straightforwardly translated into assembly. The *assembly generation* pass enforces the architectural constraints (two address instructions, forced registers). In this respect, it acts as a validation of the earlier register allocation pass. Finally, this assembly representation can be pretty-printed into usual syntax to be used by an off-the-shelf assembler or inlined into programs written in other languages (e.g., C, Rust).

5.2 Compiler correctness

The main correctness theorem of the Jasmin compiler is stated as follows. For each source program, if the compilation succeeds and produces a target program, then every execution of the source corresponds to an execution of the target. Here, execution means terminating and without errors. Formally, we have the following statement.

THEOREM 5.1 (JASMIN COMPILER CORRECTNESS).

$$\forall\, p\, p',\ compile(p) = ok(p') \rightarrow$$
$$\forall\, f,\ f \in exports(p) \rightarrow$$
$$\forall\, v_a\, m\, v_r\, m',\ enough\text{-}stack\text{-}space(f, p', m) \rightarrow$$
$$f, v_a, m \Downarrow^{p}_{call} v_r, m' \rightarrow f, v_a, m \Downarrow^{p'}_{call} v_r, m'$$

In this statement, exports(p) refers to the exported entry points of program p: non-exported functions do not appear in the target program; all calls to these functions have been inlined. Stack variables are allocated to the memory during the compilation: therefore, the compiler only preserves the semantics from initial memories in which there is enough free stack space to execute the compiled program. This property is captured by the enough-stack-space(f, p', m) predicate. Since target programs do not call functions, this predicates only states that the stack variables of the function f in the compiled program can be allocated in the initial memory m.

The conclusion of the theorem refers to the semantics of the target assembly language (x64). It is formally defined in Coq, with an accurate bit-level definition of all the instructions involved in our benchmarks.

Notice that the memories and input and output values are the same for both source and target levels: the semantics of all intermediate language share the same definitions of values and the same memory model.

Since the target language (assembly) is deterministic, for safe source programs, this theorem implies that every execution of the target corresponds to an execution of the source.

This theorem also implies that safe source programs are compiled to safe target programs, with the caveat that compiled programs may consume more stack space than the source programs. More precisely, given a source program that is safe under some precondition, the compiled program, for any initial state satisfying the precondition, will either run without run-time error, or fail by lack of stack space.

5.3 Proof methodology

The Jasmin compiler is written in OCaml and in Coq. The Coq part is formally verified, i.e., its correctness is stated and proved once and for all. The OCaml part is two-fold. On the one hand, the impure interface with the outer world (source code parsing, assembly pretty printing) is trusted.[7] On the other hand, some compilation passes call an external oracle: this oracle is written in OCaml but is not trusted; its result is validated on every run by a Coq routine that is itself verified. This lightweight proof technique, known as *translation validation* [29], is applied to many passes.

Moreover, all the validated passes only use two checkers: one is dedicated to the stack-allocation pass; the other deals with the register-allocation passes.

The checker for stack-allocation ensures that enough memory is allocated for all the local variables and that each stack access (including array accesses) is properly translated to the corresponding memory access. Its soundness relies on the safety of the source

program: the fact that all array accesses are in bounds ensures that all variables can be allocated to a single memory region.

The other validator checks that the two programs (before and after the transformation) precisely have the same structure up to some renaming. This validator is parameterized by the class of renaming that is allowed, so that it can be used in multiple passes: changes in variable names after unrolling, stack sharing and register allocation, and transformation of array indexings into variable accesses after register-array expansion.

The structure of the correctness proof follows the structure of the compiler itself: each pass is proved correct and the main theorem is a composition of these lemmas. The correctness of a compilation pass is proved through a simulation relation, which is a relational invariant of the simultaneous executions of the source and target programs.

5.4 Constant-time preservation

In addition to functional behaviour and safety, the compiler preserves constant-time security of programs. As for Jasmin programs, one can define an instrumented semantics of assembly programs, where leakage traces record the sequence of addresses accessed and program point visited during execution. One then shows that the compiler preserves constant-time: if a Jasmin program is safe and constant-time, then the generated assembly program is constant-time. Therefore, it is legitimate to prove that Jasmin programs are constant-time, using the methodology and automated tool from Section 4.

The proof of constant-time preservation for the compiler is decomposed into proving that every individual pass preserves the constant-time property. Moreover, the proof for each individual pass has a similar structure. We use a stronger statement of semantic preservation, in particular making explicit the simulation relation for this pass, and consider how this pass transforms memory accesses and branches and argue that it does so in a constant-time preserving fashion. This involves considering two executions of the target program, and distinguishing between leakages that are inherited from a similarly leaky instruction of the source program (in this case we use the hypothesis that the source program is constant-time) or are constant memory accesses (in this case no information is leaked). More formally, we rely on several crucial facts: first, compilation does not introduce branching instructions. In particular, conditional moves are compiled into CMOV instructions, and therefore have no impact on the leakage trace. Second, it is always correct to (unconditionally) remove leaks through memory accesses and branches, as they are similarly removed from all traces. This is used, e.g. to justify constant propagation and loop unrolling, which may remove (constant) branches. Third, in some cases, compilation may introduce leaky instructions. When arrays and local variables are allocated to the stack, the accesses to these variables become observable in the leakage trace; but said leakage cannot depend on secrets.

Remark. We stress that preservation of the constant-time property can be affected by aggressive compilers, which may use optimizations to tabulate expensive functions and transform constant-time computations into memory loads; or to introduce case analysis on (secret) values that are known to have a small range and

---

[7]The size of the trusted code base of Jasmin could be reduced; for instance, there are techniques for validating parsers [25]. We leave this as further work.

transform constant-time computations into branches. By design, such optimizations are not supported by the Jasmin compiler. In any case, one could argue that constant-time verification should be performed at assembly-level, eliminating the need to impose restrictions on the compiler. However, we argue that our current approach provides a reasonable compromise, with the added benefit to simplify interface with state-of-the-art verification technology.

As a final note, we observe that it would be desirable to prove preservation of constant-time using the Coq proof assistant. We leave this for future work.

5.5 Statistics

The compiler comprises about 25k lines of Coq files (not including third-party libraries), which produces about 25k lines of extracted OCaml. This certified code is complemented with 5k lines of trusted hand-written OCaml, and a few thousands lines of untrusted hand-written OCaml.

6 EVALUATION

We have evaluated the performance of compiled Jasmin programs using SUPERCOP (System for Unified Performance Evaluation Related to Cryptographic Operations and Primitives), a toolkit for measuring the performance of cryptographic software. In this work we looked at version 20170105 of the toolkit and we selected a representative set of implementations that have been hand-crafted using the qhasm language. These implementations are given as assembly programs, but they include the corresponding qhasm input program as annotations. We developed an automatic translator that starts from such assembly implementations, extracts the original qhasm code, and then automatically generates functionally equivalent Jasmin programs–for the overwhelming majority of qhasm instructions the translation is one-to-one.

Benchmarking procedure. A SUPERCOP release contains all the machinery required to carry out performance evaluation on a specific computer; any computer can be used as a representative for a particular architecture and collected data can be submitted to a central repository of collected benchmarks. In our work we have focused on execution time and we have adapted the evaluation scripts to consider specific implementations of our interest and collect high-precision clock-cycle counts. This means, in particular, that all of the implementations produced by the Jasmin compiler needed to be compliant with calling conventions imposed by GCC version 5.4.0. The machine we used is equipped with an Intel Core i7-6500U processor, clocked at 2.50 GHz, with 16 GB of RAM, running Ubuntu version 16.04. Turbo boost was disabled in all measurements. To collect clock-cycle measurements we configured SUPERCOP to use GCC -O3. (The optimization level is irrelevant for assembly implementations, and it affects equally implementations that mix C code with assembly implementations compiled from Jasmin or qhasm.) Each implementation is executed 10 000 times and the median of clock-cycle counts is taken as final result.

Benchmarking results. Table 1 displays results we collected by first measuring the speed of the original assembly implementations generated from qhasm programs, and then measuring the assembly implementations compiled from the translated Jasmin code.

Table 1: Comparison of Jasmin-generated assembly and qhasm-generated assembly. Numbers shown correspond to SUPERCOP clock-cycle counts.

Implementation	qhasm	Jasmin	Ratio
X25519-4limb-base	147 084	148 914	1.012
X25519-4limb	147 890	148 922	1.006
X25519-4limb-jasmin		143 982	
X25519-5limb-base	148 354	147 200	0.992
X25519-5limb	148 572	147 090	0.990
ed25519-5limb-keypair	55 364	56 594	1.022
ed25519-5limb-sign	83 430	85 038	1.019
ed25519-5limb-open	182 520	188 180	1.031
salsa20	12 322	12 460	1.011
salsa20-xor	12 208	12 252	1.004

Concretely, we looked at two implementations of Curve25519, for different representations of the underlying field, at the Ed25519 signature scheme [14], and at the salsa20 stream cipher. SUPERCOP testing procedures were used to ensure that both the original implementation and the new implementation are functionally correct. Our results show that, for all implementations that we have obtained using this procedure, the Jasmin toolchain essentially preserves the performance of the original implementation.

High-efficiency Jasmin. To further demonstrate that high-level programming in Jasmin is not opposite of high-efficiency, we have manually optimized the Jasmin Montgomery ladder implementation depicted in Figure 3, by carefully reordering instructions, maximizing the use of registries, and directly using the #x86_MOV(0) instruction for some variable assignments. A more detailed description of this optimized implementation is given in Appendix B. In fact, our optimized X25519-4limb-jasmin implementation[8] drops below 144k clock-cycles and beats the equivalent qhasm implementation in SUPERCOP. We believe that the same optimizations could be also performed in qhasm, but in a less modular and more expensive way, and without a proof of semantics preservation.

Benchmarking verification. All the benchmarked implementations were proved safe and constant-time at the source level using the tool presented in Section 4, and hence are guaranteed to retain these properties at the machine-level, as per the discussion in Section 5. The vast majority of qhasm-translated programs are written in a straight-line style and were automatically proven, given a two-line top-level specification of valid memory regions (safety) and public inputs (constant-time). Verification is more effective for Jasmin-style code with high-level control flow structures, especially loops, but may require suitable programmer-supplied annotations typical of high-level functional correctness proofs. This is the case of some salsa20 fragments containing loops that manipulate memory regions, and our X25519-4limb-jasmin implementation (Figure 5).

---

[8]Benchmarked using the -nolea flag to disable load effective address instructions.

7 RELATED WORK

Verification of Curve25519. There have been several efforts to reason formally about the functional correctness of Curve25519.

Chen et al. [19] prove functional correctness of a qhasm implementation of scalar multiplication. In their approach, functional correctness is captured by a post-condition stating that the program computes its intended result. Moreover, the verification process requires the programmer to annotate the qhasm code with assertions that establish relations between relevant intermediate values in the code. For instance, applying their approach to modular multiplication requires the programmer to insert an assertion at the program point where multiplication was completed, and insert subsequent assertions referring to the intermediate value at various points in the modular reduction step. Starting from a qhasm program with sufficiently many annotations, their approach generates a set of proof goals, which are automatically translated to the Boolector SMT solver [30] that attempts to discharge them automatically. When the SMT solver is not powerful enough to complete the proof that these intermediate results suffice to imply the post-condition, then the proof is completed in Coq. Intuitively, moving to Coq is necessary whenever there is a semantic gap—in this case due to complex algebraic arguments—between the expressed post-condition (e.g. $x = y \times x \mod 2p$) and the assertions that can be proved by the SMT solver, which typically encode details of how the modular reduction is carried out within the program (e.g., by first reducing modulo 2^{256} and then adjusting the result). This approach has the important benefit of reasoning directly over assembly code. However, it is labour-intensive, and delivers weak guarantees, since the qhasm language has no formal semantics.

Zinzindohoué and co-workers [36] use an approach based on refinement types for verifying functional correctness of an implementation of Curve25519 written in a stateful, verification-aware, higher-order functional programming language from the ML family. However, the assembly code is very inefficient. In a recent work, Bhargavan and co-workers [16, 35] propose a different approach for generating efficient and functionally verified C implementations; however, the fastest implementations are obtained using an unverified compiler. In an independent work, Erbsen et al. [22] propose yet another alternative approach for synthesizing functionally correct and efficient implementations from high-level specifications written in Coq. Finally, Bernstein and Schwabe [15] develop an automated tool for proving functional correctness of a C implementation of Curve25519. Starting from a sufficiently-annotated C implementation, the gfverif tool generates a set of algebraic equalities which are sufficient to guarantee functional correctness and can be proved automatically using a symbolic computation tool. All these approaches yield strong guarantees on the source programs, but the assembly code is generated with untrusted tools, which is clearly undesirable for high-assurance software. Most of these approaches also depart significantly from current practices, and require programmers to adopt non-conventional languages, which may be a serious obstacle to adoption. In addition, none of these works consider side-channel security of the assembly implementations—side-channel security of C implementations is discussed in [16], but obviates the security gap in modern compilers [21].

Other work. Our work is also closely related to Vale [17], which leverages the Dafny verifier to provide a framework for proving functional correctness and side-channel resistance of high-performance assembly code. The Vale language provides high-level control-flow structures that simplify the writing and verification of cryptographic routines. In contrast to Jasmin, the Vale compiler is not verified: all verification is performed on the generated annotated assembly.

Almeida et al. [2, 3] propose a general methodology for obtaining strong guarantees for assembly-level implementations, through proving simultaneously the three properties, using a C implementation of MEE-CBC as an illustrative case study. Their approach relies on a combination of multiple tools, including EasyCrypt for proving security of algorithmic descriptions, Frama-C for proving functional equivalence between algorithmic descriptions and C implementations, CompCert for proving functional equivalence between C implementations and assembly code, and a formally verified type system for constant-time for side-channel security.

Appel [5] leverages the Verified Software Toolchain [6] to prove functional correctness of an assembly-level implementation of SHA256 generated using the CompCert compiler [28]. In a related effort, Beringer and co-workers [10] further leverage the Foundational Cryptographic Framework [31] to prove, in addition to functional correctness, cryptographic security of an assembly-level implementation of HMAC. Recently, Ye and co-workers [34] have extended this approach for proving correctness and cryptographic security of the mbedTLS implementation of HMAC-DRGB.

Beyond these works on validating cryptographic implementations, there is a significant amount of work on building verified compilers and formal models of assembly languages.

There has been a significant amount of work on analyzing side-channel resistance of cryptographic implementations. Our work is most closely related to static analyses for cryptographic constant-time, including [8, 33], and specially to the product-based approach of ct-verif [4]. However, ct-verif targets LLVM intermediate representation, leaving open the question of carrying the results of the analysis to assembly code, while we target code that is significantly closer to assembly, and (informally) argue that the Jasmin compiler preserves cryptographic constant-time.

8 CONCLUSION

Jasmin is a framework for building high-speed and high-assurance cryptographic implementations using a programming language that simultaneously guarantees control on the generated assembly and verifiability of the source programs. We justify our design with proofs that the Jasmin compiler preserves behavior, safety, and constant-time security of source programs; the main correctness result—semantics preservation—is formally verified in the Coq proof assistant.

Our main pending task is proving functional correctness of Jasmin programs. We are completing a foundational (i.e. formally verified in Coq) infrastructure for proving correctness of Jasmin programs, and intend to leverage prior work on certified tactics for arithmetic to achieve higher automation. Another task is to build a foundational infrastructure for proving functional equivalence between two Jasmin implementations. We plan to use these tools

in combination for proving functional correctness of our Jasmin implementation of Curve25519. Moreover, we intend to include support for richer instruction sets, and for different architectures. As a first step, we intend to add support for vector instructions that are routinely used in cryptographic implementations.

REFERENCES

[1] Nadhem J. AlFardan and Kenneth G. Paterson. 2013. Lucky Thirteen: Breaking the TLS and DTLS Record Protocols. In *IEEE Symposium on Security and Privacy, SP 2013*. IEEE Computer Society, 526–540.

[2] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, and François Dupressoir. 2013. Certified computer-aided cryptography: efficient provably secure machine code from high-level implementations. In *ACM CCS 13*, Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung (Eds.). ACM Press, 1217–1230.

[3] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, and François Dupressoir. 2016. Verifiable Side-Channel Security of Cryptographic Implementations: Constant-Time MEE-CBC. In *FSE 2016 (LNCS)*, Thomas Peyrin (Ed.), Vol. 9783. Springer, Heidelberg, 163–184. https://doi.org/10.1007/978-3-662-52993-5_9

[4] Jose Carlos Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. 2016. Verifying Constant-time Implementations. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, Austin, TX. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/almeida

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

[6] Andrew W Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. 2014. *Program logics for certified compilers*. Cambridge University Press.

[7] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. 2005. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures (Lecture Notes in Computer Science)*, Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever (Eds.), Vol. 4111. Springer, 364–387. https://doi.org/10.1007/11804192_17

[8] Gilles Barthe, Gustavo Betarte, Juan Diego Campo, Carlos Daniel Luna, and David Pichardie. 2014. System-level Non-interference for Constant-time Cryptography. In *ACM CCS 14*, Gail-Joon Ahn, Moti Yung, and Ninghui Li (Eds.). ACM Press, 1267–1279.

[9] Evmorfia-Iro Bartzia and Pierre-Yves Strub. 2014. A Formal Library for Elliptic Curves in the Coq Proof Assistant. In *Interactive Theorem Proving - 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings (Lecture Notes in Computer Science)*, Gerwin Klein and Ruben Gamboa (Eds.), Vol. 8558. Springer, 77–92. https://doi.org/10.1007/978-3-319-08970-6_6

[10] Lennart Beringer, Adam Petcher, Katherine Q. Ye, and Andrew W. Appel. 2015. Verified Correctness and Security of OpenSSL HMAC. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015.*, Jaeyeon Jung and Thorsten Holz (Eds.). USENIX Association, 207–221. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/beringer

[11] Dan Bernstein. Writing high-speed software. (????). http://cr.yp.to/qhasm.html

[12] Daniel J. Bernstein. 2005. Cache-timing attacks on AES. (2005). http://cr.yp.to/antiforgery/cachetiming-20050414.pdf.

[13] Daniel J. Bernstein. 2006. Curve25519: New Diffie-Hellman Speed Records. In *PKC 2006 (LNCS)*, Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin (Eds.), Vol. 3958. Springer, Heidelberg, 207–228.

[14] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. 2011. High-Speed High-Security Signatures. In *CHES 2011 (LNCS)*, Bart Preneel and Tsuyoshi Takagi (Eds.), Vol. 6917. Springer, Heidelberg, 124–142.

[15] Dan Berstein and Peter Schwabe. 2015. gfverif: fast and easy verification of finite-field arithmetic. (2015). http://gfverif.cryptojedi.org/

[16] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Catalin Hritcu, Jonathan Protzenko, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy, Peng Wang, Santiago Zanella Béguelin, and Jean Karim Zinzindohoué. 2017. Verified Low-Level Programming Embedded in F. *CoRR* abs/1703.00053 (2017). http://arxiv.org/abs/1703.00053

[17] Barry Bond, Chris Hawblitzel, Manos Kapritsos, K. Rustan M. Leino, Jacob R. Lorch, Bryan Parno, Ashay Rane, Srinath Setty, and Laure Thompson. 2017. Vale: Verifying High-Performance Cryptographic Assembly Code. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/bond

[18] Billy Bob Brumley, Manuel Barbosa, Dan Page, and Frederik Vercauteren. 2012. Practical Realisation and Elimination of an ECC-Related Software Bug Attack. In *CT-RSA 2012 (LNCS)*, Orr Dunkelman (Ed.), Vol. 7178. Springer, Heidelberg, 171–186.

[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. 2014. Verifying Curve25519 Software. In *ACM CCS 14*, Gail-Joon Ahn, Moti Yung, and Ninghui Li (Eds.). ACM Press, 299–309.

[20] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings (Lecture Notes in Computer Science)*, C. R. Ramakrishnan and Jakob Rehof (Eds.), Vol. 4963. Springer, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[21] Vijay D'Silva, Mathias Payer, and Dawn Xiaodong Song. 2015. The Correctness-Security Gap in Compiler Optimization. In *2015 IEEE Symposium on Security and Privacy Workshops, SPW 2015, San Jose, CA, USA, May 21-22, 2015*. IEEE Computer Society, 73–87. https://doi.org/10.1109/SPW.2015.33

[22] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. 2017. Systematic Synthesis of Elliptic Curve Cryptography Implementations. (2017). https://people.csail.mit.edu/jgross/personal-website/papers/2017-fiat-crypto-pldi-draft.pdf

[23] Shay Gueron and Vlad Krasnov. 2013. The fragility of AES-GCM authentication algorithm. Cryptology ePrint Archive, Report 2013/157. (2013). http://eprint.iacr.org/2013/157.

[24] Darrel Hankerson, Alfred Menezes, and Scott Vanstone. 2004. Guide to elliptic curve cryptography. (2004).

[25] Jacques-Henri Jourdan, François Pottier, and Xavier Leroy. 2012. Validating LR(1) Parsers. In *European Symposium on Programming (ESOP)*. Springer, 397–416.

[26] Thierry Kaufmann, Hervé Pelletier, Serge Vaudenay, and Karine Villegas. 2016. When Constant-Time Source Yields Variable-Time Binary: Exploiting Curve25519-donna Built with MSVC 2015. In *Cryptology and Network Security - 15th International Conference, CANS 2016, Milan, Italy, November 14-16, 2016, Proceedings (Lecture Notes in Computer Science)*, Sara Foresti and Giuseppe Persiano (Eds.), Vol. 10052. 573–582. https://doi.org/10.1007/978-3-319-48965-0_36

[27] K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers (Lecture Notes in Computer Science)*, Edmund M. Clarke and Andrei Voronkov (Eds.), Vol. 6355. Springer, 348–370. https://doi.org/10.1007/978-3-642-17511-4_20

[28] Xavier Leroy. 2006. 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*. ACM, 42–54.

[29] George C. Necula. Translation validation for an optimizing compiler. In *ACM sigplan notices* (2000), Vol. 35. ACM, 83–94.

[30] Aina Niemetz, Mathias Preiner, and Armin Biere. 2014 (published 2015). Boolector 2.0 system description. *Journal on Satisfiability, Boolean Modeling and Computation* 9 (2014 (published 2015)), 53–58.

[31] Adam Petcher and Greg Morrisett. 2015. The Foundational Cryptography Framework. In *Principles of Security and Trust - 4th International Conference, POST 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings (Lecture Notes in Computer Science)*, Riccardo Focardi and Andrew C. Myers (Eds.), Vol. 9036. Springer, 53–72. https://doi.org/10.1007/978-3-662-46666-7_4

[32] Massimiliano Poletto and Vivek Sarkar. 1999. Linear scan register allocation. 21, 5 (1999), 895–913.

[33] Bruno Rodrigues, Fernando Magno Quintão Pereira, and Diego F. Aranha. 2016. Sparse representation of implicit flows with applications to side-channel detection. In *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, Ayal Zaks and Manuel V. Hermenegildo (Eds.). ACM, 110–120. https://doi.org/10.1145/2892208.2892230

[34] Katherine Ye, Matthew Green, Naphat Sanguansin, Lennart Beringer, Adam Petcher, and Andrew W. Appel. 2017. Verified correctness and security of mbedTLS HMAC-DRBG. In *ACM CCS 2017*.

[35] Jean Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. 2017. HACL*: A Verified Modern Cryptographic Library. *IACR Cryptology ePrint Archive* 2017 (2017), 536. http://eprint.iacr.org/2017/536

[36] Jean Karim Zinzindohoué, Evmorfia-Iro Bartzia, and Karthikeyan Bhargavan. 2016. A Verified Extensible Library of Elliptic Curves. In *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016*. IEEE Computer Society, 296–309. https://doi.org/10.1109/CSF.2016.28

A JASMIN SYNTAX REFERENCE

This section presents the concrete syntax of Jasmin source programs using BNF notation. Terminals are typeset in capital letters or literally when no confusion should arise, non-terminals are surrounded by angle brackets. Optional parts are written within square brackets. Repeated parts are suffixed by an asterisk * (not to be confused with the terminal asterisk *).

Some rules (not shown) enable to concisely express common patterns as non-empty sequences of elements (X) separated by commas (⟨tuple1⟩ X), etc.

Types. Jasmin types are boolean, mathematical integers, bit-vectors of some predetermined sizes, or arrays of such bit-vectors. The sizes of these arrays are given through arbitrary expression which must reduce to positive integers at compile-time.

$$
\begin{array}{lcl}
\langle\text{ptype}\rangle & ::= & \text{T\_BOOL} \quad | \quad \text{T\_INT} \\
& | & \langle\text{utype}\rangle \quad | \quad \langle\text{utype}\rangle \langle\text{brackets}\langle\text{pexpr}\rangle\rangle \\[4pt]
\langle\text{utype}\rangle & ::= & \text{T\_U8} \quad | \quad \text{T\_U16} \quad | \quad \text{T\_U32} \\
& | & \text{T\_U64} \quad | \quad \text{T\_U128} \quad | \quad \text{T\_U256}
\end{array}
$$

Expressions. Jasmin expressions are made of variables, array accesses, literal constants (booleans or mathematical integers), memory accesses, prefix unary operators, infix binary operators, and function and primitive calls.

Operators include usual arithmetic, boolean and bit-wise operations. When relevant, arithmetic operators come with a *signed* variant (with an 's' suffix), which interpret their arguments as signed integer.

The prefixed type in memory accesses corresponds to the type of the value to fetch; it defaults to b_{64}. The pointer expression is made of a base (variable) and an offset (expression).

$$
\begin{array}{lcl}
\langle\text{pexpr}\rangle & ::= & \langle\text{var}\rangle \\
& | & \langle\text{var}\rangle \langle\text{brackets}\langle\text{pexpr}\rangle\rangle \\
& | & \text{TRUE} \quad | \quad \text{FALSE} \quad | \quad \text{INT} \\
& | & [\langle\text{parens}\langle\text{ptype}\rangle\rangle] \langle\text{brackets}(\langle\text{var}\rangle + \langle\text{pexpr}\rangle)\rangle \\
& | & \langle\text{peop1}\rangle \langle\text{pexpr}\rangle \\
& | & \langle\text{pexpr}\rangle \langle\text{peop2}\rangle \langle\text{pexpr}\rangle \\
& | & \langle\text{parens}\langle\text{pexpr}\rangle\rangle \\
& | & \langle\text{var}\rangle \langle\text{parens\_tuple}\langle\text{pexpr}\rangle\rangle \\
& | & \langle\text{prim}\rangle \langle\text{parens\_tuple}\langle\text{pexpr}\rangle\rangle \\[6pt]
\langle\text{ident}\rangle & ::= & \text{NID} \\[6pt]
\langle\text{var}\rangle & ::= & \langle\text{ident}\rangle \\[6pt]
\langle\text{prim}\rangle & ::= & \#\,\langle\text{ident}\rangle \\[6pt]
\langle\text{peop1}\rangle & ::= & ! \quad | \quad - \\[6pt]
\langle\text{peop2}\rangle & ::= & + \quad | \quad - \quad | \quad * \\
& | & \&\& \quad | \quad \text{PIPEPIPE} \\
& | & \& \quad | \quad \text{PIPE} \quad | \quad \char`\^ \quad | \quad << \quad | \quad >> \quad | \quad >>s \\
& | & == \quad | \quad != \quad | \quad < \quad | \quad <= \quad | \quad > \quad | \quad >= \\
& | & <s \quad | \quad <=s \quad | \quad >s \quad | \quad >=s
\end{array}
$$

Instructions. A Jasmin instruction is either an array initialization, a parallel assignment (maybe conditional), a function call, a conditional branch, a for loop with explicit direction, or a while loop.

The while loop is slightly non-standard as its body is split in two parts: the first part is executed on every iteration before the condition is evaluated; the second part is executed on every iteration after the condition is evaluated (unless the condition evaluates to false, in which case the execution of the loop terminates). This enables to handle, with a single syntactic construct, usual while loops, do-while loops, and while loops whose conditions are instructions (the first part of the body) rather than simple expressions.

A sequence of instructions surrounded by braces makes a block.

$$
\begin{array}{lcl}
\langle\text{pinstr}\rangle & ::= & \text{ARRAYINIT} \langle\text{parens}\langle\text{var}\rangle\rangle \;; \\
& | & \langle\text{tuple1}\langle\text{plvalue}\rangle\rangle \langle\text{peqop}\rangle \langle\text{pexpr}\rangle \, [\text{IF}\, \langle\text{pexpr}\rangle]\;; \\
& | & \langle\text{var}\rangle \langle\text{parens\_tuple}\langle\text{pexpr}\rangle\rangle \;; \\
& | & \text{IF}\, \langle\text{pexpr}\rangle \langle\text{pblock}\rangle \\
& | & \text{IF}\, \langle\text{pexpr}\rangle \langle\text{pblock}\rangle \text{ ELSE } \langle\text{pblock}\rangle \\
& | & \text{FOR}\, \langle\text{var}\rangle = \langle\text{pexpr}\rangle \text{ TO } \langle\text{pexpr}\rangle \langle\text{pblock}\rangle \\
& | & \text{FOR}\, \langle\text{var}\rangle = \langle\text{pexpr}\rangle \text{ DOWNTO } \langle\text{pexpr}\rangle \langle\text{pblock}\rangle \\
& | & \text{WHILE}\, [\langle\text{pblock}\rangle] \langle\text{parens}\langle\text{pexpr}\rangle\rangle [\langle\text{pblock}\rangle] \\[6pt]
\langle\text{pblock}\rangle & ::= & \langle\text{braces}\langle\text{pinstr}\rangle^*\rangle
\end{array}
$$

The assignment operators are either raw or compound with a binary (arithmetic or bit-wise) operator.

$$
\begin{array}{lcl}
\langle\text{peqop}\rangle & ::= & = \\
& | & += \quad | \quad -= \quad | \quad *= \\
& | & >>= \quad | \quad >>s= \quad | \quad <<= \\
& | & \&= \quad | \quad \char`\^= \quad | \quad \text{PIPEEQ}
\end{array}
$$

Left-values (destinations of assignments) are either an underscore meaning that the value should be ignored (not assigned to anything), a variable, an array cell, or a memory address.

⟨plvalue⟩ ::= UNDERSCORE
       | ⟨var⟩
       | ⟨var⟩ ⟨brackets⟨pexpr⟩⟩
       | [⟨parens⟨ptype⟩⟩] ⟨brackets(⟨var⟩ + ⟨pexpr⟩)⟩

Functions. A function body is, surrounded by braces, a sequence (maybe empty) of declarations of local variables, followed by a sequence (maybe empty) of instructions, followed by an optional return clause. Functions may return several values at once.

⟨pfunbody⟩ ::= LBRACE (⟨pvardecl⟩ ;)* ⟨pinstr⟩* [RETURN
             ⟨tuple⟨var⟩⟩ ;] RBRACE

⟨storage⟩ ::= REG | STACK | INLINE

⟨stor_type⟩ ::= ⟨storage⟩ ⟨ptype⟩

⟨pvardecl⟩ ::= ⟨stor_type⟩ ⟨var⟩

Global declarations. A Jasmin module is a sequence of global declarations, each of them being the declaration of a function, of a parameter (value known at compile time) or of a global (read-only) variable.

⟨module⟩ ::= ⟨top⟩* EOF | error

  ⟨top⟩ ::= ⟨pfundef⟩ | ⟨pparam⟩ | ⟨pglobal⟩

⟨call_conv⟩ ::= EXPORT | INLINE

⟨pfundef⟩ ::= [⟨call_conv⟩] FN ⟨ident⟩
         ⟨parens_tuple(⟨stor_type⟩ ⟨var⟩)⟩ [->
         ⟨tuple⟨stor_type⟩⟩] ⟨pfunbody⟩

⟨pparam⟩ ::= PARAM ⟨ptype⟩ ⟨ident⟩ = ⟨pexpr⟩ ;

⟨pglobal⟩ ::= ⟨ident⟩ = ⟨pexpr⟩ ;

B JASMIN X25519-4LIMB IMPLEMENTATION

This section provides more detailed information regarding our optimized X25519-4limb-jasmin implementation in Jasmin. Figure 6 shows the complete source code for the iterated_square and mladder procedures – that exemplify the use of high-level control flow structures in Jasmin – including the respective programmer annotations needed for automatic verification. The remaining procedures consist of simple straight-line code and only require top-level procedure contract annotations. Our full X25519-4limb-jasmin implementation comprises a total of 16 procedures and 798 lines of code. These include 24 lines of programmer annotations split into procedure contracts (13 lines) and loop invariants (11 lines).

Figure 6: Complete iterated_square and mladder procedures from our X25519-4limb-jasmin implementation.

```
fn iterated_square(stack b64[4] xa, stack b64 n) ⟶ stack b64[4]
//@ requires n >=s 3 && n <=s 98;
//@ security requires public(n);
{
    reg b64[8] z; reg b64[4] r; reg b64[5] t;
    reg b64 xa0, xa1, xa2, rax, rdx;
    reg bool cf;
    reg b64 n_r;

    //@ cf = false;
    while
    //@ decreases n;
    //@ invariant !cf == (n >=s 0);
    //@ invariant n <=s 98;
    {
        xa0 = xa[0]; xa1 = xa[1]; xa2 = xa[2];

        rax = xa1; rdx, rax = rax * xa0;
        z[1] = rax; z[2] = rdx;

        rax = xa2; rdx, rax = rax * xa1;
        z[3] = rax; z[4] = rdx;

        rax = xa[3]; rdx, rax = rax * xa2;
        z[5] = rax; z[6] = rdx; z[7] = #x86_MOV(0);

        rax = xa[2]; rdx, rax = rax * xa0;
        cf, z[2] += rax; cf, z[3] += rdx + cf; _, z[4] += 0 + cf;

        rax = xa[3]; rdx, rax = rax * xa1;
        cf, z[4] += rax; cf, z[5] += rdx + cf; _, z[6] += 0 + cf;

        rax = xa[3]; rdx, rax = rax * xa0;
        cf, z[3] += rax;
        cf, z[4] += rdx + cf; cf, z[5] += 0 + cf; cf, z[6] += 0 + cf; _, z[7] += 0 + cf;

        cf, z[1] += z[1];
        cf, z[2] += z[2] + cf; cf, z[3] += z[3] + cf; cf, z[4] += z[4] + cf;
        cf, z[5] += z[5] + cf; cf, z[6] += z[6] + cf; cf, z[7] += z[7] + cf;

        rax = xa0; rdx, rax = rax * xa0;
        z[0] = rax; t[0] = rdx;

        rax = xa1; rdx, rax = rax * xa1;
        t[1] = rax; t[2] = rdx;

        rax = xa[2]; rdx, rax = rax * xa[2];
        t[3] = rax; t[4] = rdx;

        cf, z[1] += t[0];
        cf, z[2] += t[1] + cf; cf, z[3] += t[2] + cf; cf, z[4] += t[3] + cf;
        cf, z[5] += t[4] + cf; cf, z[6] += 0 + cf; _, z[7] += 0 + cf;

        rax = xa[3]; rdx, rax = rax * xa[3];
        cf, z[6] += rax; _, z[7] += rdx + cf;

        r = reduce(z);
        xa = r;

        n_r = n;
        cf, n_r -= 1;
        n = n_r;
    } (! cf)

    return xa;
}
```

```
fn mladder(stack b64[4] x2, stack b64[4] z2, stack b64[4] xr, reg b64 sp)
    ⟶ stack b64[4], stack b64[4]
//@ requires valid(sp, 8 * 0, 8 * 4 - 1);
//@ security requires public(sp);
{
    stack b64 s;
    reg b64 tmp1, tmp2, bit, swap;
    stack b64 prevbit;
    stack b64[4] x1;
    reg b64[4] x2r;
    stack b64[4] x3, z3;
    reg b64 i, j;
    stack b64 is, js;
    reg bool cf;
    reg b64[4] buf;

    buf = xr; x1 = buf; x3 = buf;
    x2[0] = 1;
    x2[1] = #x86_MOV(0);
    x2[2] = #x86_MOV(0);
    x2[3] = #x86_MOV(0);
    z2[0] = #x86_MOV(0);
    z2[1] = #x86_MOV(0);
    z2[2] = #x86_MOV(0);
    z2[3] = #x86_MOV(0);
    z3[0] = 1;
    z3[1] = #x86_MOV(0);
    z3[2] = #x86_MOV(0);
    z3[3] = #x86_MOV(0);

    j = 62; i = 3; prevbit = #x86_MOV(0);

    while
    //@ decreases i;
    //@ invariant i <=s 4;
    //@ invariant j == 62 || j == 63;
    //@ invariant (i >=s 0) ==> valid(sp, 8 * i, 8 * i + 7);
    //@ security invariant public(i);
    {
        tmp1 = [sp + 8 * i];
        is = i;
        s = tmp1;
        while
        //@ decreases j;
        //@ invariant j <=s 63;
        {
            tmp2 = s;
            bit = tmp2 >> j;
            js = j;
            bit = bit & 1;
            swap = prevbit;
            swap ^= bit;
            prevbit = bit;
            x2r, z2, x3, z3 = cswap(x2, z2, x3, z3, swap);
            x2r, z2, x3, z3 = ladderstep(x1, x2r, z2, x3, z3);
            x2 = x2r;
            j = js;
            j -= 1;
        } (j >=s 0)
        j = 63;
        i = is;
        i -= 1;
    } (i >=s 0)

    return x2, z2;
}
```