Efficient Verification of Optimized Code
Correct High-speed X25519

Marc Schoolderman[1,2], Jonathan Moerman[1],
Sjaak Smetsers[1], and Marko van Eekelen[1,2]

[1] Radboud University, Nijmegen, The Netherlands
{m.schoolderman, jmoerman, s.smetsers, marko}@science.ru.nl
[2] Open University of the Netherlands, Heerlen, The Netherlands
{marc.schoolderman, marko.vaneekelen}@ou.nl

Abstract. Code that is highly optimized poses a problem for program-level verification: programmers can employ various clever tricks that are non-trivial to reason about. For cryptography on low-power devices, it is nonetheless crucial that implementations be functionally correct, secure, and efficient. These are usually crafted in hand-optimized machine code that eschew conventional control flow as much as possible.
We have formally verified such code: a library which implements elliptic curve cryptography on 8-bit AVR microcontrollers. The chosen implementation is the most efficient currently known for this microarchitecture. It consists of over 3000 lines of assembly instructions.
Building on earlier work, we use the Why3 platform to model the code and prove verification conditions, using automated provers.
We expect the approach to be re-usable and adaptable, and it allows for validation. Furthermore, an error in the original implementation was found and corrected, at the same time reducing its memory footprint.
This shows that practical verification of cutting-edge code is not only possible, but can in fact add to its efficiency—and is clearly necessary.

1 Introduction

Although formal verification is considered to give the highest level of assurance in security-critical software [21], it is seldom applied. Even if a verification technique is expressive enough to reason about a given problem domain, for its use to make economic sense, it must be usable by programmers proficient in that domain, and not require an excessive amount of time. These criteria are hard to meet.

Cryptographic implementations are always security-critical: subtle bugs can have disastrous consequences [9], and the security of a system is only as strong as its weakest link. As Chen et al. [10] note, the desire to avoid risk in cryptographic implementations can hamper adoption of new and more efficient crypto libraries, simply because the correctness of these implementations cannot be properly demonstrated. As they also note, a full audit in addition to testing can be extremely expensive, and impractical for high-performance implementations due to extensive use of clever optimizations. In this context the case of

applying formal verification looks very reasonable, and indeed this is actively pursued [7, 10].

However, this poses many important challenges. First, at what level should verification occur? Compilers have been known to be a source of concern, as they can cause subtle problems [22]. Second, understanding the formal verification process used can be a daunting task: powerful tools such as the Verification Software Toolchain [1] have a substantial learning curve. If instead an ad-hoc method is used, the correctness of the method itself needs to be clearly established for it to be trustworthy. Lastly, cryptography by its nature involves the exploitation of carefully engineered mathematics, which a formal method must be able to state and work with, which adds to the effort required in showing correctness of implementations.

To rely on the verification of any code—cryptographic or otherwise—its specification must be validated as well. This demands a formal specification that is succinct, and comprehensible by a domain expert. Furthermore, we do not want to decide between efficiency and correctness: both are important, and in fact verification ideally assists in making implementations more efficient. Finally, for a verification technique to be practical, it should be re-usable for other verification tasks in the future, and not simply a one-shot operation.

In this paper, we present such a technique, by applying the existing Why3 verification platform [17] to prove the functional correctness of a highly optimized library used for X25519 elliptic curve cryptography on 8-bit microcontrollers [14]. We arrive at a succinct specification, and we expect our technique to be capable of verifying similar code for more powerful processors with less effort.

1.1 Contributions

We provide a corrected version of an X25519 implementation optimized for the 8-bit AVR architecture. Our modifications, described in Section 6, improve upon the fastest implementation currently known for this challenging architecture [14].

We demonstrate functional correctness and memory safety of this implementation by providing a machine-checked proof using the Why3 verification platform [17]. Concretely, we prove that the code calculates a scalar multiplication on Curve25519 by applying a double-and-add scheme—the *Montgomery ladder*—using Montgomery's x-coordinate-only formulas [27].

We also provide a formal Why3 model of a subset of the AVR instruction set, that has been carefully constructed for easy validation with respect to the official specification [2]. This model can be re-used for other purposes, or modified to fit a different verification purpose without loss of its validity.

We describe our approach in using Why3 for this verification task; this is an extension of earlier work [30], and has been demonstrated to have a low barrier to understanding [31]. This approach should work similarly well for other architectures such as ARM or RISC-V. The overall methodology is not specific to the domain of cryptographic implementations.

1.2 Availability of Results

The code belonging to this paper is available online in an open repository.[3] To check the proofs, Why3 version 0.88.3 is required.[4] For discharging the verification conditions the provers CVC3 (2.4.1), CVC4 (1.4 and 1.6), Z3 (4.6.0), and E-prover (2.0) were used.

2 Elliptic Curve Cryptography on Small Devices

X25519 is a public key cryptography scheme built around a Diffie-Hellman key exchange [5, 24]. 'Original' Diffie-Hellman obtains its security through the observation that, given a primitive root g for a prime p, it is (in general) hard to compute g^{xy} from g^x and g^y (mod p) without knowing the integers x or y [12]. For proper security, a sufficiently large prime modulus p is needed—2048 bits is a recommended minimum [33]. Performing the required exponentiation and modular reduction steps on such large integers is hard to do efficiently on restricted devices [20]. Also, the viability of side-channel attacks prescribes various precautions on all code that computes using secret data, to ensure that an implementation does not inadvertently leak information [18].

2.1 Curve25519

Using elliptic curves eases some of these issues [5]. Given a field $\mathbb{F}$, and coefficients $A, B \in \mathbb{F}$, a *Montgomery curve over* $\mathbb{F}$ is defined as all the points $x, y \in \mathbb{F}$ that satisfy the formula:

$$By^2 = x^3 + Ax^2 + x$$

To this set of points is added a 'point at infinity' denoted $\mathcal{O}$ to form an additive group. When P, Q are *distinct* points on the curve, $P + Q$ is defined as the third point on the curve that intersects the straight line passing through P and Q, reflected around the x-axis. For $P + P$, the tangent of the curve at point P is used to find this point. The point at infinity $\mathcal{O}$ acts as the neutral element.

The separate cases of point *adding* and *doubling*, can be used to compute a *scalar multiple* $n \cdot P$, or P added to itself n times, using a double-and-add scheme. Again a *Diffie-Hellman assumption* [23] applies: if $\mathbb{F}$ is a finite field of prime order, it is assumed to be hard to compute $nm \cdot P$ from $n \cdot P$ and $m \cdot P$.

X25519 performs a scalar multiplication on Curve25519: a Montgomery curve over the finite field $\mathbb{F}_p$ where $p = 2^{255} - 19$, and coefficients $A = 486662, B = 1$. The choice of p facilitates efficient modular reductions. Furthermore, Montgomery [27] gives efficient formulas for both *doubling* and *differential addition* of points, which only requires the x-coordinates of points. These formulas derive their efficiency by representing an x-coordinate by the ratio $X : Z$, with $x \equiv X \cdot Z^{-1} \bmod p$.

[3] https://doi.org/10.5281/zenodo.4640377
[4] Later versions do not yet support our approach—see Section 8

The scalar multiple $n \cdot P$, finally, is computed using the *Montgomery ladder*. This can be mathematically described by the following formula:

$$\textsc{ladder}\ n\ P = \begin{cases} (\mathcal{O},\ P) & \text{if } n = 0 \\ (2R_0,\ R_1 + R_0) & \text{if } n > 0 \text{ and even} \\ (R_1 + R_0,\ 2R_1) & \text{if } n > 0 \text{ and odd} \end{cases}$$
$$\text{where in the last two cases } (R_0,\ R_1) = \textsc{ladder}\ \lfloor n/2 \rfloor\ P$$

It can be shown that for every $n \geq 0$, $\textsc{ladder}\ n\ P = (n \cdot P,\ (n+1)P)$, but instead of computing $n \cdot P$ using a naive double-and-add scheme, this definition performs the same arithmetic operations in both recursive cases — the only difference between the recursive cases is a swap of the arguments. This enables a constant-time implementation [6].

2.2 X25519 on AVR

The AVR microarchitecture is an 8-bit RISC architecture [2], and so we can only represent an element $x \in \mathbb{F}_p$ by splitting it into 32 bytes. Since the AVR only has 32 registers (of which some are needed as index registers), no single element $x \in \mathbb{F}_p$ can be loaded from memory entirely. Therefore, judicious register allocation is of prime concern for an efficient implementation. Therefore, all of the primitives operations in $\mathbb{F}_p$ are rendered in assembly code in [14]. These comprise the following:

- A $256 \to 256$-bit routine subtracting $2^{255} - 19$ from its input (with borrow).
- A $256 \times 256 \to 512$-bit multiplication routine, constructed by recursive application of Karatsuba's algorithm out of smaller $32 \times 32 \to 64$-bit multiplication routines.
- A $256 \to 512$-bit dedicated squaring routine of similar construction
- A $512 \to 256$-bit modular reduction function, which given a $m \in \mathbb{F}_p$ computes $\hat{m}$ so that $\hat{m} \equiv m \pmod{p}$ and $\hat{m} < 2^{256}$, used to reduce the results of the previous two functions.
- $256 \times 256 \to 256$-bit modular addition/subtraction routines which perform a multi-precision addition/subtraction with a built-in modular reduction.
- A specialized $256 \to 256$-bit routine for efficient modular multiplication with the constant 121666.

Other operations are rendered in C code: these are either very simple, or consist mostly of function calls to these primitive operations. Examples of such functions would be a $256 \times 256 \to 256$-bit modular multiplication, a function that canonicalizes an element $x \in \mathbb{F}_p$ by repeated subtraction of p, and a function that computes $x^{-1} \pmod{p}$ using Fermat's little theorem.

The Montgomery ladder is implemented in C iteratively as illustrated by Algorithm 1. Essentially this computes the scalar multiple using the same double-and-add scheme as the Ladder function defined above, starting at the most significant bit of its input, and swapping the roles of $(X_1 : Z_1)$ and $(X_2 : Z_2)$ as

needed. We will show in Section 5.1 that the informal specification given here is *not* entirely correct. The LADDERSTEP procedure shown in Algorithm 1 is an optimized implementation of Montgomery's formulas [27] for doubling and adding points. Note that the literature usually only presents the Montgomery ladder in this iterated version, often—confusingly—with minor variations to the LADDERSTEP procedure [5, 24]. We find this optimized form of the Montgomery ladder hard to understand, making its full formal verification desirable.

Algorithm 1 Montgomery ladder for scalar multiplication

Require: A 255-bit scalar n, and a x-coordinate x_P of a point P
Ensure: Result $(X:Z)$ satisfies $x_{n\cdot P} \equiv X \cdot Z^{-1}$

$(X_1:Z_1) \leftarrow (1:0)$; $(X_2:Z_2) \leftarrow (x_P:1)$; $prev \leftarrow 0$; $j \leftarrow 6$
for $i \leftarrow 31$ **downto** 0 **do**
  while $j \geq 0$ **do**
    $bit \leftarrow$ **bit** $8i + j$ **of** n
    $swap \leftarrow bit \oplus prev$; $prev \leftarrow bit$
    if swap **then** $(X_1:Z_1, X_2:Z_2) \leftarrow (X_2:Z_2, X_1:Z_1)$ ▷ by conditional moves
    LADDERSTEP$(x_P, X_1 : Z_1, X_2:Z_2)$
    $j \leftarrow j - 1$
  end while
  $j \leftarrow 7$
end for
return $(X_1:Z_1)$

procedure LADDERSTEP
  $T_1 \leftarrow X_2 + Z_2$
  $X_2 \leftarrow X_2 - Z_2$
  $Z_2 \leftarrow X_1 + Z_1$
  $X_1 \leftarrow X_1 - Z_1$
  $T_1 \leftarrow T_1 \cdot X_1$
  $X_2 \leftarrow X_2 \cdot Z_2$
  $Z_2 \leftarrow (Z_2)^2$
  $X_1 \leftarrow (X_1)^2$
  $T_2 \leftarrow Z_2 - X_1$
  $Z_1 \leftarrow T_2 \cdot 121666$
  $Z_1 \leftarrow Z_1 + X_1$
  $Z_1 \leftarrow T_2 \cdot Z_1$
  $X_1 \leftarrow Z_2 \cdot X_1$
  $Z_2 \leftarrow T_1 - X_2$
  $Z_2 \leftarrow (Z_2)^2$
  $Z_2 \leftarrow Z_2 \cdot x_P$
  $X_2 \leftarrow T_1 + X_2$
  $X_2 \leftarrow (X_2)^2$
end procedure

3 Why3 Verification Platform

Why3 [17] is a verification platform for deductive program verification. It comprises the typed programming language WhyML (which can be annotated with functional contracts and assertions), as well as libraries for reasoning about specific types of objects (such as arrays, bit-vectors, bounded and unbounded integers), which the user can also extend. A weakest-precondition calculus generates the correctness condition for an annotated program, which Why3 then

transforms into the input language for various automated or interactive provers. Besides assertions and contracts, WhyML also provides other means of instrumenting programs to aid verification. We highlight two:

Ghost code is guaranteed by the type system to not have any effect on the actual execution on the code, but can be used to compute witnesses for use in verification goals.

Abstract blocks can be used to summarize multiple operations with a single functional contract.

An advantage of Why3's reliance on automatic provers is that verification does not need to be the last step in a waterfall-like process. When a program (or specification) is changed, most of the verification conditions that held previously can usually be solved again at the press of a button, even when the change affects them. Similarly, if a prover can solve one instance of a problem, it can usually—given enough time—handle similar or larger instances, allowing for proofs to be transplanted. For instance, we recycled parts of the proofs of [30]. Since Why3 uses multiple provers in concert, we are not restricted by the limitations of one particular (version of) a prover. In this sense, proofs seem robust.

On the other hand, too much irrelevant information can hinder automatic provers. Sometimes an assertion frustrates a proof that is completely unrelated to it. In this sense, proofs can also be brittle. Thus, for large verification tasks keeping the proof context small is vitally important. We used Why3's module system, *ghost code* and *abstract blocks* to keep the proof context manageable.

4 Correctness of Low-level Code

In the implementation we are considering, all primitives for implementing the field arithmetic needed for computing in $\mathbb{F}_p$ are implemented in assembly code. With the exception of the multiplication routine, this code is free of conditional branches. In the multiplication routine, branches are used, but in every case, both branches take the same amount of clock cycles and perform the same sequence of memory accesses. This should prevent a side-channel attack such as described by Genkin et al. [18], which exploits observed timing differences. Our formal verification effort therefore only focuses on the functional correctness and memory-safety of these routines.

Since 256-bit operations are not natively supported on any CPU, an X25519 implementation usually chooses a representation where an element $x \in \mathbb{F}_p$ is represented in n *limbs* in radix 2^w; that is, $x = \sum_{i=0}^{n} 2^{iw} x_{[i]}$ for the limbs $x_{[0]}, x_{[1]}, \ldots x_{[n-1]}$. If these limbs can contain more than w bits of information, this representation is called *unpacked*, and any carry that occurs during computation does not need to be propagated to the next limb immediately. An *unpacked* representation with few limbs is more efficient, and is thought to be more convenient for verification [10, 35]. On the implementation for the AVR a *packed* representation of 32 limbs in radix 2^8 is used, and every part of the code is forced to handle carry-propagation.

Globally, our approach follows that of [30]; we specify the representation of a 256-bit multi-precision integer in terms of an 8-bit memory model, model every AVR mnemonic that is needed as a WhyML function, and mechanically translate the assembly code to this model for verification with Why3.

4.1 A Re-usable Validated AVR Machine Model

For modeling the processor state, we use the concept of an *8-bit address space*, which is a Why3 `map` of addresses to integers, suitably restricted:

```
type address_space = { mutable data: map int int }
  invariant { forall i. 0 <= self.data[i] < 256 }
```

The AVR register file, data segment, and stack are all modeled as separate address spaces. This of course means that our model is an underspecification, but most assembly code conforms to this simplified model. Memory size restrictions are not part of the definition of an *address space*, as it is more convenient to express them as pre-conditions for the AVR instructions that manipulate memory. To model the carry and 'bit transfer' CPU flags, we use the equivalent of a `ref bool`; the value of all other flags are unspecified. We also use *ghost registers* [30] to track register updates inside abstract blocks using Why3's type system.

Since we needed to model many AVR instructions, we first implemented (in WhyML) a *primitive instruction set* of common operations on these *address spaces*, such as reading and writing 8-bit and 16-bit values represented either by their integer value, or as bit-vectors. These operations are verified for consistency with the *8-bit address space*. This instruction set is then used to *implement* all required AVR instructions following the official specification [2].

For example, for the `SUBI` instruction, the AVR specification tells us that a constant K will be subtracted from its destination register, and the carry flag will be set to $\overline{r_7} \cdot K_7 + K_7 \cdot r'_7 + r'_7 \cdot \overline{r_7}$ (in boolean arithmetic), where x_7 denotes the most significant bit of an 8-bit value x, and r, r' are the previous and updated values of the destination register, respectively. In terms of our primitives, we can state this as:

```
let subi (rd: register) (k: int)
  requires { 0 <= k <= 255 }
= let rdv  = read_byte reg rd in
  let res  = clip (rdv - k) in
  set_byte reg rd res;
  cf.value <- (not ar_nth rdv 7 && ar_nth k 7 ||
               ar_nth k 7 && ar_nth res 7 ||
               ar_nth res 7 && not ar_nth rdv 7)
```

While this follows the official specification closely, it is not very useful for verifying programs. Capturing the common notion that the carry flag gets set if and only if $r < K$ can be done by adding a Why3 contract for `subi`:

```
ensures { reg = old reg[rd <- mod (old (reg[rd] - k)) 256] }
ensures { ?cf = -div (old (reg[rd] - k)) 256 }
```

That is, the register file gets updated with the destination register receiving $(r - K)$ mod 256, and the numeric value of the carry flag will be $-\lfloor \frac{r-K}{256} \rfloor$.

Why3 allows us to verify that this contract is satisfied by the AVR specification.[5] Also, if a different contract were discovered to be more useful, it could easily be replaced while maintaining validity of the model.

Extensions to the model Some of the code verified featured a limited form of branching. We modeled this using a WhyML function that throws an exception if the branch is taken; this exception is then handled at the appropriate location.

In two locations, data on the stack was allocated for use with memory operations, which our simplified model did not support. We resolved this by adding the requirement that the stack pointer does not alias with any of the ordinary data inputs, and checking manually whether the code conforms to the conventions for accessing memory on the stack. As we will explain in Section 6.2, this turned out not to be the case, necessitating modifications.

4.2 Proving the correctness of AVR assembly code

For all of the assembly routines, we of course want to show *functional correctness*. However, since these routines must interface with C code, we also have to verify that they are well-behaved. This means proving that they only modify the memory that they are allowed to (i.e. temporary data on the stack or that passed by the caller as a pointer), that they leave the stack in a consistent state, and that they adhere to the C calling convention for the AVR [19].

Note that there are two versions of the 256-bit multiplication routines in [14]: one which uses function calls to the respective 128-bit operation, and one which inlines everything for a very minor increase in speed. We consider the former to be the more relevant one, and so have chosen that as our verification target.

Quantitative verification results are shown in Table 1. The vast majority of the goals were discharged by CVC3 and CVC4. The number of annotations required gives a *rough* measure of the manual effort. This is a subjective number since not every annotation represents the same amount of effort. As a point of reference, verifying `fe25519_mul121666` was measured to take 16 hours of work.

Verification by partitioning into blocks The 256×256-bit multiplication is constructed by using calls to a 128×128-bit multiplication routine using Karatsuba's method. The 128×128-bit multiplication routine itself, is comprised of three in-line applications of a 64×64-bit Karatsuba multiplication, the basic version of which was verified earlier in [30]. Some parts of this earlier proof could in fact simply be re-used.

[5] For `SUBI`, this also revealed a mistake in online documentation.

function	*instructions*	*user annotations*	*generated goals*	*CPU time*
`bigint_mul256:mul128`	1078	122	300	1504.6s
`bigint_mul256`	693	85	506	2000.1s
`bigint_square256:sqr128`	672	26	135	363.8s
`bigint_square256`	493	38	359	1796.6s
`bigint_subp`	103	12	84	184.0s
`fe25519_red`	305	41	182	155.3s
`fe25519_add`	242	52	209	156.4s
`fe25519_sub`	242	53	212	119.6s
`fe25519_mul121666`	138	56	149	393.0s

Table 1. Results of verifying the X25519 field arithmetic in AVR assembly

For the 128-bit and 256-bit larger versions, the proofs followed a similar approach, with one notable change. For the smaller Karatsuba routines, it sufficed to identify 7 'blocks' of code, and state their operations in *contextual terms*—i.e., specifying which part of Karatsuba's algorithm each block performed. For more than one level of Karatsuba, this becomes unwieldy. While we kept the identified blocks the same, we found it much more useful—even for routines verified in [30]—to specify their effects in purely *local* terms—i.e, only specifying what its effect is in terms of its immediately preceding state. For some blocks, this simplifies the specification, and actually makes the work for automatic provers slightly easier. In cases where this contextual information *is* required, it can always be re-asserted later. The only drawback we have found to this method was that on assembly code of this size, it is easy to lose sight of what one is trying to achieve without reliable contextual information.

The 256-bit squaring routine is similarly constructed out of calls to a 128-bit squaring routine; both compute the square of $A = 2^w A_h + A_l$ as $A^2 = (2^w+1)(2^w A_h^2 + A_l^2) - 2^w(A_l - A_h)^2$, which we are able to verify by partitioning these routines into 5 blocks.

Instrumenting programs with ghost code The routines that perform modular arithmetic are very different in style from the multiplication routines. In the latter, we can apply a decomposition into a small number of large blocks, which allows SMT solvers to do most of the work. The reduction, addition and subtraction routines, by contrast, are highly repetitive—essentially the same read-modify-write sequence repeated several times.

In this case, it was more logical to use a bottom-up approach, summarizing the effects of these short sequences using a WhyML function (essentially the same idea as using an assembly *macro*), which is then iterated. We discovered, however, that after a few macro applications, SMT solvers were unable to prove memory safety or absence of aliasing. The culprit here seemed to be that the macros accessed memory via `LD+`/`ST+` instructions (which perform a load/store, followed by a pointer increment). Perhaps unsurprisingly, it becomes increasingly hard for SMT solvers to reason about out what address an index register is referring to after many modifications have been applied to it.

In our routines (and we suspect, commonly in similar cases) such addresses are however perfectly obvious, and can be statically deduced. We therefore instrumented the code with *ghost arguments*, which supply this missing information. As a simple example of this technique (which was also used in the 256×256-bit multiplication routine), we can make the AVR `LD+` instruction (modeled as the WhyML function `AVRint.ld_inc`) more amenable to verification by instrumenting it with ghost arguments:

```
let ld_inc' (dst src: register) (ghost addr: int)
  ... (* the specification of AVRint.ld_inc *)
  requires { uint 2 reg src = addr }
= AVRint.ld_inc dst src
```

On the surface, this just appears to add a needless pre-condition; however, once this correlation between `addr` and `uint 2 reg src` is established, SMT solvers can use this information to easily deduce what address the index register used is referring to.

5 Correctness of the C Code

The X25519 implementation we verify also consists of around 300 lines of C code, which interfaces directly with the assembly routines verified in Section 4. Many routines are short and simple, and verification for them is a straight-forward application of Why3.

To ensure that the C code and the assembly code are both verified with respect to the same logical foundations, we translate C by hand into the WhyML primitives from Section 4.1, that underpin the AVR instruction set model. However, since a C compiler handles allocation of global and local variables, using one `address_space` to model memory would be impractical and incorrect, as it would force the model to make assumptions about the memory layout. So instead, every array object is modeled as residing in its own `address_space`. An added benefit of this is that Why3's type system will enforce that arguments do not alias. The minor drawback is that some functions can be called to perform in-place updates, which does requires aliasing. These functions have to be modeled and verified for both cases separately.

For the assembly routines that interface with the C code, abstract specifications are added by duplicating the contracts of the verified assembly routines, and removing the pre- and post-conditions related to the C calling conventions.

The verification results are listed in Table 2. Among the field operations, it is notable that `fe25519_unpack` and `fe25519_invert` generate more goals. The former is due to its (RFC-required) bit-masking of its input, which we specify as a reduction mod 2^{255}. We suspect our proof of this function can be further optimized, but decided against spending time on this. Note that the field arithmetic code actually operates on a *packed* representation, so `unpack` and `pack` functions are otherwise simply copy-operations.

The `fe25519_invert` function computes $x^{2^{255}-21} \pmod{p}$ using sequences of modular square-and-multiply steps. This makes it very similar to repetitive assembly code, and it is treated the same way: we instrument the code with *ghost arguments* in a highly regular fashion which specify the actual value of intermediate results—which interestingly was more or less a formalization of the *inline comments* provided by the original authors. Also, *abstract blocks* helped keep the number of verification conditions small.

For the verification of the last three routines in Table 2, verification was 'simply' an effort of finding the correct invariants and assertions that guided the automatic provers to the desired conclusion within an acceptable amount of CPU time. To achieve the final conclusion presented in Section 5.2, it is required to know that 2^{255-19} is a prime number; we took the pragmatic route and stated this as an axiom in Why3. To see that $x^{2^{255}-21}$ is the multiplicative inverse of x also requires Fermat's little theorem, which we instead proved inside Why3 using *ghost code* that traces a direct proof using modular arithmetic—showing that for any integer a not divisible by a prime p, it is the case that $a^{p-1}\prod_{i=1}^{p-1} i \equiv \prod_{i=1}^{p-1} a \cdot i \equiv \prod_{i=1}^{p-1} i$, and therefore $a^{p-1} \equiv 1$.

function	*lines*	*user annotations*	*generated goals*	*CPU time*
`fe25519_setzero`	3	2	7	0.4s
`fe25519_setone`	4	2	7	0.4s
`fe25519_neg`	3	0	3	0.2s
`fe25519_cmov`	5	3	10	36.5s
`fe25519_freeze`	4	2	9	4.7s
`fe25519_unpack`	4	8	30	41.0s
`fe25519_pack`	5	2	11	1.6s
`fe25519_mul`	3	0	1	0.2s
`fe25519_square`	3	0	1	0.1s
`fe25519_invert`	51	49	306	557.3s
`work_cswap`	8	0	13	3.8s
`ladderstep`	26	22	80	202.8s
`mladder`	26	22	140	345.1s
`crypto_scalar_mult_curve25519`	13	27	57	74.2s

Table 2. Results of verifying the X25519 C routines

5.1 Verifying the Montgomery Ladder

Montgomery [27] provides formulas for doubling and differential addition of points on an elliptic curve, where only the x-coordinates of these points on the curve are used. As mentioned in Section 2.1, these x-coordinates are represented as *ratios* $(X : Z)$, where $x \equiv X \cdot Z^{-1} \bmod p$. The point at infinity $\mathcal{O}$, which is not on the curve, is represented by $(X : Z)$ with $X \neq 0, Z = 0$. The degenerate case $(0 : 0)$ does not represent anything.

For Curve25519, Montgomery's formulas are proven correct for all cases by Bernstein [5], and look as follows:

$$X_{2n} = (X_n^2 - Z_n^2)^2 \qquad X_{m+n} = 4Z_{m-n}(X_m X_n - Z_m Z_n)^2$$
$$Z_{2n} = 4X_n Z_n (X_n^2 + 486662 X_n Z_n + Z_n^2) \qquad Z_{m+n} = 4X_{m-n}(X_m Z_n - Z_m X_n)^2$$

If the x-coordinate of the point nP is the ratio $(X_n : Z_n)$, then $(X_{2n} : Z_{2n})$ is the ratio for the point $(2n)P$. Likewise, from x_{nP} and x_{mP}, we can compute $x_{(m+n)P}$ provided we also know $x_{(m-n)P}$.

We have proven that the `ladderstep` procedure (see Algorithm 1), given values $(x, X_n : Z_n, X_m : Z_m)$, computes $(X_{2n} : Z_{2n}, X_{m+n} : Z_{m+n})$ as specified by these point doubling and addition formulas, with $X_{m-n} = x$, and $Z_{m-n} = 1$.

To verify the function `mladder` (Algorithm 1), we define a formal specification in Why3 of the Montgomery ladder as presented in Section 2.1, but using the above formulas for doubling and addition. We verify that `mladder` adheres to this specification: if for some 255-bit integer s and x-coordinate x_P, LADDER s $(x_P : 1)$ returns $(X : Z)$ as the first component of its result, `mladder` computes $(\tilde{X} : \tilde{Z})$ such that $\tilde{X} \equiv X$ and $\tilde{Z} \equiv Z \pmod{p}$.

Importantly, for this result to hold, we found it necessary to require that s is even, and has its most significant bit set. The former is necessary, as an odd s would leave the results of Algorithm 1 in a state where a final swap is still needed. Having bit 254 in s set is necessary, as it prevents Algorithm 1 from performing the doubling formula on the 'point at infinity', which would make it impossible to demonstrate the strict correspondence.

These requirements on s are however taken care of by the existence of the 'clamping' operation in X25519, which requires $s \in \{2^{254} + 8k : 0 \leq k < 2^{251}\}$. Having s a multiple of 8 is crucial for the mathematical security of X25519 [24]. Setting the high bit is done for entirely different reasons: to prevent programmers from applying a non-constant-time optimization that reveals information about the scalar s [23]. Our formal proof was greatly helped by this choice, perhaps providing more justification for it.

5.2 A Succinct Specification of X25519

The function `crypto_scalar_mult` is our ultimate verification goal. We show the most important part of the specification proven in Why3 here:

```
val crypto_scalarmult_curve25519 (r s p: address_space)
  ensures { uint 32 r = mod (uint 32 r) p25519 }
  ensures { let xp   = mod (uint 32 p) (pow2 255) in
            let mult = scale (clamp (uint 32 s)) xp in
            if mult ~ infty then
              uint 32 r === 0
            else
              uint 32 r ==~ mult }
```

Informally, the first post-condition states that the result is in canonical form, i.e. fully reduced. The second post-condition states that, after the high bit of the x-coordinate of P is masked (as per RFC7748 [24]), a ratio $(X : Z)$ representing the x-coordinate of $[s] \cdot P$ is computed using repeated application of Montgomery's formulas (where $[s]$ is the clamped value of s). If $[s] \cdot P$ happens to be $\mathcal{O}$, the function writes a zero result; otherwise the result will be equivalent to $x_{[s] \cdot P}$.

Note that is not possible to distinguish the result $[s] \cdot P = \mathcal{O}$ and $x_{[s]P} = 0$. However, for every point P whose y-coordinate is not-zero, X25519 also does not distinguish P and $-P$; this specification elucidates that $\mathcal{O}$ and the point at the origin $(x = 0, y = 0)$ are similarly unified.

6 Improved X25519 for AVR

Several small improvements were observed, which we confirmed by a formal proof. Two instructions in the 128×128-bit multiplication assembly routine could be removed with no impact on the formal proof, confirming they were unnecessary. In `fe25519_freeze`, the routine `bigint_subp` is called twice to fully reduce an integer mod $2^{255} - 19$. We were able to verify that one call suffices, since in the current implementation it is always applied to a result that is already partially reduced.

6.1 Memory Safety

In [30], several version of the Karatsuba implementations could compute incorrect results if the memory locations used for storing input and output were aliased, so we were naturally curious about aliasing in the X25519 implementation. We found that the prohibition on aliasing also applies to the 128-bit and 256-bit multiplication/squaring routines,[6] and the `fe25519_red` modular reduction function. The modular addition/subtraction routines and `fe25519_mul121666` were verified to be safe when used for in-place update operations.

The C code calls all these functions accordingly, so aliasing never becomes an issue. We did add a `restrict` keyword to the function prototypes for which argument aliasing results in undefined behavior.

6.2 Interrupt Safety

The 256-bit multiplication and squaring routines use function calls to the 128-bit versions to compute their results, which expect their arguments to be in memory. One of these calls multiplies an intermediate result and so has to write this back to memory using the stack.

However, the original code did this by writing the data below the stack pointer. This means that if the microcontroller is interrupted importunely (e.g.

[6] As a peculiar exception: the 128-bit squaring routine will function properly when reading from address i and writing to address $i + 8$

due to a timer or I/O event), and an associated interrupt service routine needs this stack space for local variables, this data is clobbered. The problem can be demonstrated by forcing an interrupt.

This problem was discovered during the modeling phase of verification, as our initial AVR model needed an extension to support direct access to the stack pointer, forcing us to consider the conditions under which this is allowed. We replaced the faulty code with code that moves the stack pointer using an idiomatic sequence [3], which we added to our model. Due to our formal proof, we were also able to see that in the 256×256-bit multiplication some of the memory reserved for the final output was available for use as a temporary, reducing the amount of total stack space required by 32 bytes.

7 Related Work

Verified cryptography has gained much interest. In [35], a verified library of elliptic curves written in F* is presented. These provide the foundation for the C implementation of X25519 in the HACL* library [34]: an implementation is created in a intermediate language Low*, verified against the F* specifications, and then mechanically translated into C. EverCrypt [29] includes a similar C implementation, as well as an efficient implementation in x86-64 assembly code, which is similarly generated, but using the Vale [8] tool. Vale is essentially a high-level assembly language with support for deductive reasoning, with a focus on cryptographic applications. A similar X25519 implementation, now included in BoringSSL [15], uses Coq [11] to generate efficient C code. All these approaches involve *generation* of *new* implementations.

Efforts to verify *existing* full implementations also exist. In [13], an ECDSA implementation in Java is proven equivalent with a Cryptol [16] specification. This is also a partially automated proof, requiring 1500 lines of annotation guiding the proof (in the form of SAWScript). Compared to our approach, the Cryptol specification is less succinct—it actually is a complete, low-level implementation in its own right, written in a functional language.

Recently, the X25519 implementation in TweetNaCl has been verified [32] using Coq and VST [1]. This implementation was, however, designed with verification in mind. The proof states that TweetNaCl (when compiled with CompCert [25]) correctly implements a scalar multiplication. Like [35], the authors show this with respect to a formal *mathematical* specification of elliptic curves.

Two efficient X25519 implementations written in 64-bit `qhasm` were partially verified by Chen et al [10]. Their approach is comparable to ours, in that they generate verification conditions which they solve using Boolector. However, where we use Why3 for this, they uses a custom approach, and report lengthier verification times. Their verification is partial, in the sense that they show that their Montgomery ladderstep implementation matches that of Algorithm 1, but don't verify the ladder itself. Similarly, Liu et al. [26] have verified several C routines of OpenSSL by compiling them to the LLVM intermediate representation, and translating that to the dedicated verification language CryptoLine.

8 Conclusion

To our knowledge, our result is the first to fully verify an existing high-speed implementation of X25519 scalar multiplication, and the first to present a verified implementation optimized for low-power devices. We show correctness with respect to short formulas that are themselves proven correct in the literature [5,27].

Like [32], only general purpose, well-understood verification methods were used. Why3 in particular has an easy learning curve [31]. Our method for translating C and assembly code into WhyML is straight-forward, and the AVR model of Section 4.1 can be validated, so trust in our results mainly resides with trusting the verification condition generation of Why3, the soundness of the automated provers, and the compilation-toolchain (C compiler, assembler and linker) used for producing AVR binaries. The weakest link in this chain is definitely the use of automated provers: during our work we discovered a soundness error in Alt-Ergo 2.0, forcing us to preclude its use. We eagerly await the ability to perform proof reconstruction in Why3 using verified SMT solvers [4, 28].

We used a version of Why3 compatible with [30]. Newer versions are available, which in principle allow for an improved AVR model and specification. However, due to a change in the meaning of *type invariants*, the versions available to us generated inefficient SMT output for the verified multiplication routines of [30]. Since our use of type invariants can be avoided, we explored several alternatives, but in the end chose to use the older version for time-efficiency reasons.

Our verification was performed in an amount of time that seems commensurate with the time it took the original implementers to engineer the code. Most time was spent on the multiplication routines in assembly code. For the C code, the most time-consuming part was, in fact, finding the right abstraction level for a simple specification of the Montgomery ladder.

Due to our general purpose approach, our findings are encouraging for other low-level language applications. In particular, due to the limitations of AVR, the code we encountered was quite long, and performed arithmetic on many *limbs* (32 instead of the more usual four or five). We expect our approach to work well for verifying the 32-bit ARM code in [14], requiring less time and with the possibility of some proof re-use. We would also like to verify the compiler-generated assembly code of routines verified at a higher level (such as in Section 5), by translating high-level specifications to the assembly level. This would strengthen our result by removing the C compiler from the trusted code base.

Acknowledgments The authors thank Benoît Viguier and Peter Schwabe for their advice, as well as the anonymous reviewers for their comments. This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Agreement No. HR.00112090028. This work is part of the research programme 'Sovereign' with project number 14319 which is (partly) financed by the Netherlands Organisation for Scientific Research (NWO).

References

1. Appel, A.W.: Verified Software Toolchain. In: Barthe, G. (ed.) Programming Languages and Systems. pp. 1–17. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
2. Atmel Corporation: AVR Instruction Set Manual, revision 0856L (2016)
3. AVR Libc Project: avr-libc User Manual, https://www.nongnu.org/avr-libc/user-manual/FAQ.html
4. Barbosa, H., Blanchette, J.C., Fleury, M., Fontaine, P., Schurr, H.J.: Better SMT Proofs for Easier Reconstruction. In: AITP 2019 - 4th Conference on Artificial Intelligence and Theorem Proving. Obergurgl, Austria (Apr 2019)
5. Bernstein, D.J.: Curve25519: New Diffie-Hellman speed records. In: Yung, M., Dodis, Y., Kiayias, A., Malkin, T. (eds.) Public Key Cryptography - PKC 2006. pp. 207–228. Springer Berlin Heidelberg, Berlin, Heidelberg (2006)
6. Bernstein, D., Lange, T.: Montgomery curves and the Montgomery ladder. Cryptology ePrint Archive, IACR (2017)
7. Bhargavan, K., et al.: Everest: Towards a Verified, Drop-in Replacement of HTTPS. In: Lerner, B.S., Bodík, R., Krishnamurthi, S. (eds.) 2nd Summit on Advances in Programming Languages (SNAPL 2017). Leibniz International Proceedings in Informatics (LIPIcs), vol. 71, pp. 1:1–1:12. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2017)
8. Bond, B., Hawblitzel, C., Kapritsos, M., Leino, K.R.M., Lorch, J.R., Parno, B., Rane, A., Setty, S., Thompson, L.: Vale: Verifying high-performance cryptographic assembly code. In: Proceedings of the 26th USENIX Conference on Security Symposium. pp. 917–934 (2017)
9. Brumley, B.B., Barbosa, M., Page, D., Vercauteren, F.: Practical realisation and elimination of an ECC-related software bug attack. In: Dunkelman, O. (ed.) Topics in Cryptology – CT-RSA 2012. pp. 171–186. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
10. Chen, Y.F., Hsu, C.H., Lin, H.H., Schwabe, P., Tsai, M.H., Wang, B.Y., Yang, B.Y., Yang, S.Y.: Verifying Curve25519 software. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. p. 299–309. CCS '14, Association for Computing Machinery, New York, NY, USA (2014)
11. The Coq proof assistant reference manual (2015), https://coq.inria.fr/documentation
12. Diffie, W., Hellman, M.: New directions in cryptography. IEEE Trans. Inf. Theor. **22**(6), 644–654 (Nov 1976)
13. Dockins, R., Foltzer, A., Hendrix, J., Huffman, B., McNamee, D., Tomb, A.: Constructing semantic models of programs with the Software Analysis Workbench. In: Blazy, S., Chechik, M. (eds.) Verified Software. Theories, Tools, and Experiments. pp. 56–72. Springer International Publishing, Cham (2016)
14. Düll, M., Haase, B., Hinterwälder, G., Hutter, M., Paar, C., Sánchez, A.H., Schwabe, P.: High-speed Curve25519 on 8-bit, 16-bit, and 32-bit microcontrollers. Des. Codes Cryptography **77**(2–3), 493–514 (Dec 2015)
15. Erbsen, A., Philipoom, J., Gross, J., Sloan, R., Chlipala, A.: Simple high-level code for cryptographic arithmetic - with proofs, without compromises. In: 2019 IEEE Symposium on Security and Privacy (SP). pp. 1202–1219 (2019)
16. Erkök, L., Carlsson, M., Wick, A.: Hardware/software co-verification of cryptographic algorithms using Cryptol. In: 2009 Formal Methods in Computer-Aided Design. pp. 188–191 (2009). https://doi.org/10.1109/FMCAD.2009.5351121

17. Filliâtre, J.C., Paskevich, A.: Why3 — where programs meet provers. In: Felleisen, M., Gardner, P. (eds.) Proceedings of the 22nd European Symposium on Programming. Lecture Notes in Computer Science, vol. 7792, pp. 125–128. Springer (2013)
18. Genkin, D., Valenta, L., Yarom, Y.: May the fourth be with you: A microarchitectural side channel attack on several real-world applications of Curve25519. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. p. 845–858. CCS '17, Association for Computing Machinery, New York, NY, USA (2017)
19. GNU Project: avr-gcc ABI, https://gcc.gnu.org/wiki/avr-gcc
20. Gura, N., Patel, A., Wander, A., Eberle, H., Shantz, S.C.: Comparing elliptic curve cryptography and RSA on 8-bit CPUs. In: International workshop on cryptographic hardware and embedded systems. pp. 119–132. Springer (2004)
21. ISO: ISO/IEC 15408-1:2009 Information technology—Security techniques—Evaluation criteria for IT security—Part 1: Introduction and general model (2009)
22. Kaufmann, T., Pelletier, H., Vaudenay, S., Villegas, K.: When constant-time source yields variable-time binary: Exploiting Curve25519-donna built with MSVC 2015. In: Foresti, S., Persiano, G. (eds.) Cryptology and Network Security. pp. 573–582. Springer International Publishing, Cham (2016)
23. Kleppmann, M.: Implementing Curve25519/X25519: A tutorial on elliptic curve cryptography. Tech. rep., University of Cambridge, Department of Computer Science and Technology (2020)
24. Langley, A., Hamburg, M., Turner, S.: Elliptic Curves for Security. RFC 7748 (Jan 2016), https://rfc-editor.org/rfc/rfc7748.txt
25. Leroy, X.: Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In: 33rd ACM symposium on Principles of Programming Languages. pp. 42–54. ACM Press (2006)
26. Liu, J., Shi, X., Tsai, M.H., Wang, B.Y., Yang, B.Y.: Verifying arithmetic in cryptographic c programs. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 552–564. IEEE (2019)
27. Montgomery, P.L.: Speeding the Pollard and elliptic curve methods of factorization. Mathematics of Computation **48**, 243–264 (1987)
28. de Moura, L.M., Bjørner, N.: Proofs and refutations, and Z3. In: LPAR Workshops. vol. 418, pp. 123–132. Doha, Qatar (2008)
29. Protzenko, J., Parno, B., Fromherz, A., Hawblitzel, C., Polubelova, M., Bhargavan, K., Beurdouche, B., Choi, J., Delignat-Lavaud, A., Fournet, C., et al.: Evercrypt: A fast, verified, cross-platform cryptographic provider. In: 2020 IEEE Symposium on Security and Privacy (SP). pp. 983–1002. IEEE (2020)
30. Schoolderman, M.: Verifying branch-free assembly code in Why3. In: Paskevich, A., Wies, T. (eds.) Verified Software. Theories, Tools, and Experiments. pp. 66–83. Springer International Publishing, Cham (2017)
31. Schoolderman, M., Smetsers, S., van Eekelen, M.: Is deductive program verification mature enough to be taught to software engineers? In: Proceedings of the 8th Computer Science Education Research Conference. p. 50–57. CSERC '19, Association for Computing Machinery, New York, NY, USA (2019)
32. Schwabe, P., Viguer, B., Weerweg, T., Wiedijk, F.: A Coq proof of the correctness of x25519 in TweetNaCl. In: 2021 IEEE 31th Computer Security Foundations Symposium (CSF). p. (to appear) (2021)
33. Velvindron, L., Baushke, M.D.: Increase the Secure Shell Minimum Recommended Diffie-Hellman Modulus Size to 2048 Bits. RFC 8270 (Dec 2017), https://rfc-editor.org/rfc/rfc8270.txt

34. Zinzindohoué, J.K., Bhargavan, K., Protzenko, J., Beurdouche, B.: HACL*: A Verified Modern Cryptographic Library. In: ACM Conference on Computer and Communications Security (CCS). Dallas, United States (Oct 2017)
35. Zinzindohoué, J.K., Bartzia, E., Bhargavan, K.: A verified extensible library of elliptic curves. In: 2016 IEEE 29th Computer Security Foundations Symposium (CSF). pp. 296–309 (2016)

A Formal specification of X25519 Scalar Multiplication

```
type ratio = { x: int; z: int }
constant infty: ratio = {x=1;z=0}

constant p25519: int = pow2 255 - 19
predicate (===) (x y: int) = mod x p25519 = mod y p25519
predicate (~) (p q: ratio) = x p*z q === x q*z p
predicate (==~) (x:int) (xz: ratio) = xz ~ {x=x; z=1}

function add (m n mn: ratio): ratio
  = { x = 4*z mn*sqr(x m*x n - z m*z n);
      z = 4*x mn*sqr(x m*z n - z m*x n) }

function double (n: ratio): ratio
  = { x = sqr(sqr(x n) - sqr(z n));
      z = 4*x n*z n * (sqr(x n) + 486662*x n*z n + sqr(z n)) }

function ladder (n: int) (p: ratio): (ratio, ratio)

axiom ladder_0: (*these axiomatic definitions are proven consistent*)
  forall p.ladder 0 p = ({x=1; z=0}, p)

axiom ladder_even:
  forall p, n. n > 0 -> let (r0,r1) = ladder n p in
    ladder (2*n) p = (double r0, add r1 r0 p)

axiom ladder_odd:
  forall p, n. n >= 0 -> let (r0,r1) = ladder n p in
    ladder (2*n+1) p = (add r1 r0 p, double r1)

function scale (n: int) (m: int): ratio
  = let (r,_) = ladder n {x=m; z=1} in r

function clamp (x: int): int
  = mod x (pow2 254) + pow2 254 - mod x 8

val crypto_scalarmult_curve25519 (r s p: address_space)
  ensures { uint 32 r = mod (uint 32 r) p25519 }
  ensures { let xp   = mod (uint 32 p) (pow2 255) in
            let mult = scale (clamp (uint 32 s)) xp in
            if mult ~ infty then
              uint 32 r === 0
            else
              uint 32 r ==~ mult }
```