Systematic Synthesis of Elliptic Curve Cryptography Implementations

Andres Erbsen
MIT
andreser@mit.edu

Jade Philipoom
MIT
jadep@mit.edu

Jason Gross
MIT
jgross@mit.edu

Robert Sloan
MIT
varomodt@gmail.com

Adam Chlipala
MIT
adamc@csail.mit.edu

Abstract

The cryptographic code that runs the Internet is subject to intense manual optimization by elite programmers. Most of the complexity of the optimized code comes from manipulation of integers too large to fit in hardware registers. Perhaps surprisingly, for a change as innocuous as changing an algorithmic parameter to a different prime number, significant pieces of code are rewritten from scratch. Only a handful of experts on the planet are seen as competent enough to do it well, and new implementations (which often include significant amounts of handwritten assembly) tend to take months to code and debug. In this paper, we demonstrate that the work of those experts can be automated while simultaneously increasing our confidence in code correctness. We implemented a framework in the Coq proof assistant for generating efficient code for elliptic curve cryptography (ECC), with proofs of conformance to a whiteboard-level specification in number theory. While some past projects have *verified* this kind of code, ours is the first to *synthesize* it from security parameters. We also have a smaller trusted code base than in past work, as all of our formal reasoning is done within Coq. Still, our generated code is *faster* than past work that verified clean-slate implementations. We come within a factor of 6 of the running time of the handcrafted world-champion implementations.

`/* XXX: Can it really happen that r<0?, See HAC, Alg 14.42, Step 3. If so: Handle it here!*/`
~*ed25519 "ref" implementation on SUPERCOP* [1]

1. Introduction and Existing Work

Code that implements cryptographic functionality is subject to several constraints: at the very least, it needs to have high throughput and low latency, its execution time needs to be independent of secret inputs, and it needs to use a minimal amount of system resources. In practice, it is these requirements that turn a 2^6-line executable specification into 2^{13} lines of low-level code that is only intelligible to a small group of experts. Furthermore, since all these properties need to hold even in the presence of malicious input crafted with knowledge of the code, randomized testing against a reference implementation (the current practice of the industry) is helpful but distinctly insufficient because the input space is enormous by design.

In this paper we examine elliptic curve cryptography (ECC). Today it is typical for every new TLS connection to apply ECC key agreement and signing, and connections are often short-lived enough that the cost of these cryptographic operations dominates a company's bottom-line costs to maintain a simple HTTPS website, so performance is crucial. Since these operations are security-critical, so is correctness. However, comments like the one at the beginning of this paper betray that even the most expert programmers are uncertain of the correctness of what they write.

To summarize our contribution: we have built the first framework for *synthesizing* implementations of elliptic curve cryptography. No new program code needs to be written for each new set of parameters, e.g. large prime numbers, though the state of practice today, in open-source and industry, is laborious reimplementation and testing for each prime. Our framework is a library within the Coq proof assistant, and we trust no other formal-methods tools, giving us a much *smaller trusted code base* than in past related work, which we achieve in part by formalizing more aspects of the ways that elliptic curves are used, rather than just details of arithmetic. Compared to past work on verification of ECC primitives, despite our stronger guarantees and more automated development process, our code is *much faster*, by a factor of 50. Our code is attached as a supplement to this paper.

We are aware of three substantial achievements in assuring correctness of ECC[1]. Chen et al. [16] showed that the body of the main loop of expert-written assembly code for the X25519 (then Curve25519) implementation on 64-bit processors corresponds to the "x-coordinate Montgomery ladder" implementation strategy. The verification was split between two tools: the Boolector SMT solver and Coq for goals that the former could not discharge. An overlapping set of authors later came up with a tool they describe as more lightweight and that significantly reduces the burden of verification: gfverif [10] works on C code by replacing the machine integers with symbolic variables, equations over which are solved by Gröbner-basis computation in Sage. The tool is described as an "early experiment" and an "alpha test" (and indeed, there are known bugs in Sage's Gröbner-basis computation – Sage issues 17676 and 9645). Based on their experiences, the authors identify 10 criteria that they (as crypto engineers) consider important in its design. We summarize their helpful criteria under three headings: effort (for existing code and new code), rigor (scope and soundness), and generality (of environments and computations).

In this terminology, ECC-star by Zinzindohoue et al. [35] strikes a good balance between the desired qualities: while they do require the code to be written in a new language (F*) and executed using a managed runtime, they aim to minimize effort by sharing executable code between different elliptic curves and using elaborate verification tools. Similarly to gfverif, the main achievement is verification of scalar-multiplication subroutines, but this time for three curves. However, the library uses simpler implementation techniques than the code verified in gfverif, and lags in performance by a factor of 290.

All three projects verify the most important properties of the code in question, but real-world use of these functions makes more assumptions about the properties. For example, while it is crucial that the output finite-field element is represented canonically (otherwise the redundancy would allow for an information leak), we find that the top-level specifications of none of the three projects guarantee that. Furthermore, all three top-level specifications are at a lower level than the traditional one for Internet standards. In order to avoid missing such subtle assumptions, it is necessary to verify at a higher level, a task to which existing approaches are not suited. Therefore, we set ourselves the following goals:

- Rigor: Wide enough to catch all known classes of functional-correctness bugs *and* to satisfy requirements of potential clients of the elliptic-curve library. Specs should match RFCs (the standards documents of the Internet), and the trusted code base should be small.
- Effort: Follow ECC-star in controlling the cost per set of parameters by implementing a very general library of proofs. Go further in also implementing all algorithms and their proofs parametrically, in a way that supports generating efficient low-level code while doing most reasoning in a high-level, verification-friendly language.
- Generality: Use state-of-the-art implementation strategies. Do not commit to design decisions that would rule out a known-good implementation strategy.

We will give a brief overview of context and our system's structure, then proceed to describe and evaluate our implementation.

2. Overview and Structure

From the standpoint of formal methods and correct compilation, many cryptographic algorithms have a similar flavor. They start from moderately complex number theory, based on modular arithmetic over numbers too large to fit in registers on commodity processors. To reach high-performance implementations, it is necessary to substitute more clever data structures for the natural ones of the specification, and these new structures must be proven equivalent to the originals. The heart of any efficient implementation, and the part most likely to lead to security-critical bugs even in code written by experts, is big-integer arithmetic. Traditional big-integer libraries based on dynamic allocation are hopelessly inefficient, when hand-coded assembly can be as much as 100 times faster than generic constant-time code. The main focus of the framework we have built is proof-generating synthesis of custom big-integer code, following the optimization tricks from world-champion implementations. Crucially, the expertise behind those optimizations is encapsulated once and for all in our framework, so that new variants can be generated on demand, soundly and automatically.

We refer to appendix A for additional details of elliptic curves, but a quick primer here will set the stage for understanding what follows. Elliptic curves are algebraic domains that are useful for digital signatures and other cryptographic operations. An Edwards curve with parameters a and d is defined to contain precisely the points (x, y) for which $ax^2 + y^2 = 1 + dx^2y^2$. Points, along with a point-addition operation, form a commu-

[1] We are not aware of any significant verification of other finite-field cryptography, and techniques previously applied to symmetric cryptography tend not to scale to cover the gap between our very high-level specification and low-level code.

Top-Level Equations

Elliptic-curve points uppercase, scalars in lowercase.
Juxtaposition is scalar multiplication; $3P = P + P + P$

$$sB \overset{?}{=} R + hA$$

Elliptic-Curve Operations

Code shown is point addition ((+) in the verification equation).
These coordinates are elements of a finite field, so
the (+), (−), and juxtaposition are modular arithmetic.

$$(x_1, y_1) + (x_2, y_2) = \left(\frac{x_1y_2 + x_2y_1}{1 + dx_1x_2y_1y_2}, \frac{y_1y_2 - ax_1x_2}{1 - dx_1x_2y_1y_2} \right)$$

Prime-Field Operations (Section 3)

Code shown is multiplication (x_1x_2 in the point-addition formula).
This multiplication operates on a representation of field
elements as lists of integers; see section 3.2.
The argument "base" is the base system used. For
instance, [3,7] represents $3 + (2^{26} \cdot 7)$ in base [1, 2^{26}].

```
mul (base us vs : list Z) :=
(fix mul' usr vs :=
  match usr with
  | [] => []
  | u :: usr' =>
    add  (mul' usr' vs)
         (scmul u (shift base (len usr') vs))
  end) (rev us) vs.
```

Figure 1: Abstraction layers (fully understanding the code and math here is not essential)

tative group. Most operations can be computed reasonably quickly, but inverting point-scalar multiplication is believed to be intractable and gives rise to the security of ECC.

In other words, a good abstract characterization of our domain is: We start with arithmetic modulo a large prime number. These numbers form the ingredients within a larger algebraic structure. Our goal is to support efficient computation of operations in the larger algebraic structure, while generating proof trails at compile time that relate optimized low-level code back to its relatively simple specification.

One of the key challenges is that optimizations are implemented across multiple abstraction layers. Performant low-level code includes optimizations in the representation of elliptic-curve points, in arithmetic on the big integers that make up point coordinates, and in the way those integers are distributed across smaller machine integers. Keeping all of these layers and representations in one's head is challenging but necessary for writing the straightline assembly used in practice; hence why so few people can write that code. However, using Coq, we can separate the abstraction layers for the purpose of proofs and reasoning, and then collapse the abstraction layers by performing partial evaluation at the lowest level (see section 4.1). This last step allows us to precompute operations on constants and simplify operations, speeding up our field-multiplication code by a factor of 13.

Figure 1 displays some of the abstraction layers we used. Our performance relies upon optimizations across all levels; these optimizations do not generally need to be aware of each other, and are both easier to prove and more useful in their most general forms. All the proofs on the level labeled "Elliptic Curve Operations," for example, know only that coordinates are elements of some field; they do not rely on the specific field or details of its operations. Similarly, proofs at the level of signature creation and checking (labeled "Top-Level Equations" in Figure 1) reason about an arbitrary group; they do not know that the group's elements are 2-coordinate points or what the point-addition formula is. In fact, it is crucial that those proofs not rely on the specific group implementation; there is an optimization that uses 4-coordinate points (with some redundancy) and an optimized addition formula (see section 5.2). The easiest path to proving properties of the group is to prove them about the 2-coordinate version and then prove that the 2- and 4-coordinate versions are isomorphic.

A similar strategy is employed for prime fields. Reasoning about prime fields is best done by reasoning about unbounded integers modulo a prime number. However, computers cannot represent unbounded integers, and field elements are large enough that they must be spread across several bounded machine words. Cleverness in precisely how they are spread out, however, can make a significant difference in performance by reducing data dependencies and allowing fast modular-reduction algorithms (section 3.1). These multiword implementations are quite complicated to reason about directly, so we reason about the easiest representation, the unbounded integers, and then prove an isomorphism with our efficient machine representation. Because of the generic structure of the proofs at the elliptic-curve level, we can simply swap out representations without needing new proofs.

Obtaining the multiword implementation is a challenge in itself, and forms our main accomplishment in terms of synthesis. The production-quality implementations of this functionality in open-source and industry are hardcoded for different primes and base systems, despite the general rules lurking beneath them. So, in the interests of creating as broad a library as possible,

1. Functional Operations

```
mul (base us vs : list Z) :=
(fix mul' usr vs :=
  match usr with
  | [] => []
  | u :: usr' =>
    add  (mul' usr' vs)
         (scmul u (shift base (len usr') vs))
  end) (rev us) vs.
```

2. Low-Level Operations

```
mul (f g : tuple Z 10) : tuple Z 10 :=
let fg0 := f0* g0 + 19 * (2 * f9 * g1 + f8 * g2 +...) in
let fg1 := fg0>> 26 + f1  * g0 + f0 * g1 + 19 * ... in
let fg2 := fg1>>25 + f2 * g0 + 2 * f1 * g1 +  ... in
...
let fg9 := fg8>>26 + f9 * g0 + f1 * g8 +  ... in
(fg0, fg1, fg2, ..., fg9).
```

3. Flattened with Bounded Types

```
mul (f g : tuple word64 10) :=
  let x := Word64.mul f0 g0 in
  let x0 := Word64.mul g1 2 in
  let x1 := Word64.mul f9 x0 in
  ...
  let x21 := Word64.add x1 x20 in
  let x22 := Word64.mul 19 x21 in
  let x23 := Word64.add x x22 in
  let x24 := Word64.shr x23 26 in
  ...
  (x248, x249, ... x257).
```

Figure 2: Synthesis stages for prime-field operations

we wrote prime-field operations in a maximally generic way and then synthesized concrete implementations in several stages (figure 2).

Our synthesis pipeline divides naturally into high-level and low-level transformations. Section 3 explains the high-level part, implementing modular big-integer arithmetic in terms of multiple integers at fixed precisions supported by commodity processors. In section 4 we explain how to transform the resulting functional programs into efficient low-level code that looks like assembly, where every intermediate result fits in a hardware register with a provable lack of arithmetic overflow. After explaining the synthesis pipeline for modular arithmetic, we take a step back and explain some of the idiosyncrasies of how we extended the pipeline to the complete problem of elliptic-curve signatures. We evaluate our synthesized system for security, performance, and ease of development, compare in more detail with related efforts, and conclude with thoughts on remaining barriers to deployment in production.

3. Finite-Field Operations

Prime fields, as used in cryptography, are based on the three classic operations of addition, subtraction, and multiplication. Division is defined, but efficient algorithms avoid it, so we use a slow placeholder version. It is easy enough to define their meanings in the usual style from undergraduate discrete-math classes. Commodity processors even support these operations natively for particular moduli that do not happen to be prime (e.g., 2^{64}). Still, it is surprisingly difficult to implement other moduli efficiently. Our flagship case study uses modulus $2^{255} - 19$, which is not so trivial to implement on top of 32-bit or 64-bit machine words. (Because of our strategy of creating specific code from generic proofs, we do not have to commit to a particular word size.)

3.1 Prime-Specific Modular Arithmetic

The choice of $2^{255} - 19$ for elliptic-curve cryptography is not an accident, but rather the result of a sophisticated analysis of modular-arithmetic performance [6, "Why this field?"]. Primes of the form $2^k - c$ with "small" c are called pseudo-Mersenne, and they are the first choice in cryptography due to the following extremely simple and fast modular-reduction rule:

$$\forall a, b.\ (2^k a + b) \bmod (2^k - c) = (ca + b) \bmod (2^k - c)$$

Note that just replacing $2^k a$ with ca does not necessarily produce output within the interval $[0, 2^k - c)$, but it does significantly reduce the size of the integer in question. The correctness of this rule follows from $m \bmod m = 0$ and $a + b \bmod m = (a \bmod m + b \bmod m) \bmod m$, and in particular, it does not depend on having $b < 2^k$ (even though this does lead to smaller output). Therefore, it suffices to have the big-integer implementation expose a way to efficiently *split* a number into a, b such that b is relatively small, and we never need to make the notion of "relatively small" precise. This flexible specification allows for significant performance improvements in the big-integer implementation when synthesizing the latter to a specific k.

3.2 Representing Fixed-Size Integers

We will at least follow the practice that we all learned in grade school of representing numbers according to base systems. The natural choice would be to make each digit, say, 32 bits wide, so that a representation like $[d_0, d_1, d_2]$ could stand for the number $d_0 + d_1 \cdot 2^{32} + d_2 \cdot (2^{32})^2$. We call this base system *uniform* because successive digits are associated with successive powers of a fixed base. Surprisingly, performance-competitive implementations adopt *nonuniform* base systems, where different digits are represented with different numbers of bits. We will refer to each machine word in a base system as a *limb*, and use *limb widths* to reference the weight of each integer. In the uniform 32-bit system described above, for example, the limb widths would all be 32. Given limb widths b_i, an n-limb tuple $x_0 \ldots x_{n-1}$ represents the following integer:

$$\sum_{i=0}^{n-1} x_i 2^{p_i} \text{ where } p_i = \sum_{j=0}^{i-1} b_j$$

Carrying immediately:

$$\begin{array}{cccc} & 1 & 2 & 6 \\ + & 3 & 8 & 2 \\ \hline & 5 & 0 & 8 \end{array}$$

Delayed carrying:

$$\begin{array}{cccc} & 1 & 2 & 6 \\ + & 3 & 8 & 2 \\ \hline & 4 & 10 & 8 \end{array}$$

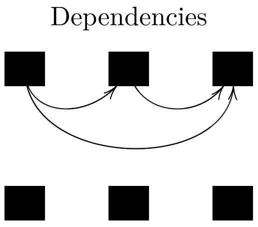

Figure 3: Comparing dependencies of traditional addition and delayed-carrying addition

The choice of limb widths for best performance is heavily influenced by k. To achieve competitive modular-reduction performance, it is necessary to implement the split operation by simply regrouping the limbs. For example, $k = 256$ implemented using 4 64-bit limbs can be split into two pairs of two words in 0 CPU cycles! However, naively dividing k into equally sized limbs would only allow implementing $255 = 3 \cdot 5 \cdot 17$ with 15-bit, 17-bit, or 51-bit words – all of which would have disappointing performance on a 32-bit platform. Thus we designed our big-integer library to operate on arbitrary nonuniform baseystem representations, for example $255 = 26+25+26+25+26+25+26+25+26+25$. We would say this representation has "limb widths" [26,25,26,25,26,25,26,25,26,25].

This approach has another significant benefit: avoiding *data dependencies* between the constituent operations, allowing processors to run several of them in parallel (see figure 3). Naively adding two 32-bit numbers can produce a 33-bit result, and handling the carry bit right away after every addition would create a chain of data dependencies between additions of different limbs. If some number of high bits of each limb are known to be zero, we can add two field elements by adding corresponding words, without carrying. In fact, we can add six 255-bit numbers before carrying, since there are 32 bits available and at most 26 are initially set – and the optimized equations we derive for curve operations never perform that many additions in a row anyway. Notably, using more limbs needs more instructions, both due to the potential explicit carries and increased number of positions, but the computation takes fewer cycles on a modern pipelined processor due to the less restrictive data dependencies.

3.2.1 Carry Chains

When not every addition is required to be followed by a carry, choosing when and which limbs to carry is critical for keeping the limb values bounded. In full generality, carrying position i consists of replacing x_i in a limb with $x_i \bmod 2^{b_i}$ and adding $\lfloor x_i/2^{b_i} \rfloor$ to the limb at the next (more significant) position. This does not change the value of the number being represented but reduces the value of the ith limb significantly at the cost of a slight increase in limb $i+1$. Carrying from the most significant limb would naively create a new limb, but if the new limb would be in the 2^k position, the carry operation can be efficiently combined with a modular reduction by adding $\lfloor x_i/2^{b_i} \rfloor c$ to x_0 instead.

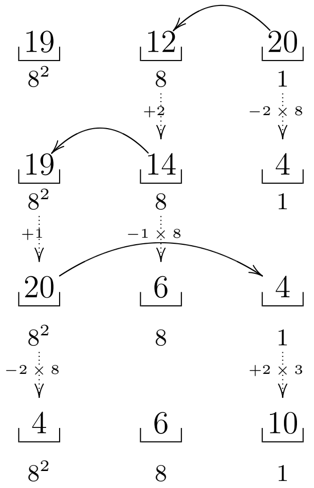

Figure 4: Carry chain (0, 1, 2) in base 8 (mod $8^3 - 3$).

A *carry chain* can be written down as an ordered list of limb numbers. For example, figure 4 represents applying the carry chain (0, 1, 2) to a three-limb number modulo $8^3 - 3$, in a uniform base-8 representation. The original number (19, 12, 20) represents $19 \cdot 8^2 + 12 \cdot 8 + 20$. We carry from limb 0 to limb 1 by observing that $12 \cdot 8 + 20 = 14 \cdot 8 + 4$, so we can represent exactly the same number with the digits (19, 14, 4). Similarly, we carry from limb 1 to limb 2 and obtain (20, 6, 4). Carrying from the third digit is more complicated, and here it becomes important that we are using modular arithmetic. If there were a fourth digit, we would use the same process as before and get the number (2, 4, 6, 4), with the 2 having weight 8^3. However, modulo $8^3 - 3$, adding $2 \cdot 8^3$ is the same as adding $2 \cdot 3$. So we can instead add $2 \cdot 3$ to the first digit, with weight 1.

Even for just the choice of performing one chain of carries right after every multiplication (rather arbitrary, but convenient for elliptic-curve arithmetic), the design space of carry chains is vast, and finding the best choices is a subject of ongoing work [11, page 9]. Instead of trying to encode all relevant constraints in our formal model, we choose to take the carry chain as a synthesis parameter. For $\mathbf{F}_{2^{255}-19}$ (the field modulo $2^{255}-19$), the one we chose for our main case study, multiple efficient carry chains are already known, and we use one of them. It would also be feasible to programmatically generate a variety of "reasonable guesses," benchmark the ones for which the entire synthesis process succeeds, and use the fastest one of those.

3.2.2 Canonical Representations

One downside of this representation is that it is redundant; that is, there are many ways to represent the same field element. This can be problematic when trying to determine if two sets of limbs represent the same field element. In these cases, it is necessary to obtain a *canonical* representation. In the case of $\mathbf{F}_{2^{255}-19}$, for example, this will be a set of limbs that evaluates to an integer in the range $[0, 2^{255} - 19)$, and in which none of the 10 limbs has more bits set than its corresponding limb width (i.e. 25 or 26 bits).

Efficiently converting an integer to its canonical representation is rather tricky. Any specific implementation consists of a fixed number of sequential carry operations where each operation carries away from the limb that the previous one carried into. Each limb needs to be carried from three times. However, proving that the resulting representation is indeed canonical required a complex invariant.

3.2.3 Correctness of Field Operations

The generic implementations and correctness proofs of multiplication, addition, subtraction, carrying, and canonicalization are the most complicated part of our development, totaling about 4000 lines of code. This, however, was a one-time cost: the specifications of operations for limbs relate them to operations on mathematical integers by stating that the two sequences of operations in the following square (for multiplication) produce the same answer for all input limbs:

$$\begin{tikzcd}
\text{limbs} \times \text{limbs} \arrow[r, "\text{mul}"] \arrow[d] & \text{limbs} \arrow[d] \\
\mathbb{Z} \times \mathbb{Z} \arrow[r, "\text{mul}"] & \mathbb{Z}
\end{tikzcd}$$

In other words, if two sets of limbs A and B represent the integers x and y respectively, then the result of our optimized multiplication on A and B would represent $x \cdot y$. This specification allows us to *reason* about field elements like unbounded integers, despite *operating on* a much more complicated representation.

Concretely, what we get from this phase are executable Coq functions for the different operations, parameterized over lists of digit multipliers for nonuniform base systems. (See, for instance, the `mul` function in Figure 2.) Calling these functions naively, we recurse through dynamically allocated lists of digits, simultaneously doing lookups in dynamically allocated lists of digit multipliers. In other words, we still face much of the runtime overhead of traditional big-integer libraries.

4. Synthesizing Low-Level Code

Though the high-level transformations provide the heart of the clever algorithms we will employ in the end, it still remains to avoid the runtime overheads of running functional programs directly. Our goal is proof-generating transformation to functional programs that look just like assembly code: sequences of `let` bindings of native integers. The first challenge is to flatten the *control flow* of the high-level programs, simplifying away all uses of lists using knowledge of the precise parameters we are synthesizing for. The next challenge is to analyze the *dataflow* sufficiently well to prove that all high-level operations on unbounded integers are accurately simulated with machine integers.

This buys us a factor of about 100 in running time.

4.1 Partial Evaluation

The purpose of partial evaluation in our pipeline is to flatten all the carefully crafted abstraction layers introduced earlier in the process. Our general, recursive multiplication function becomes a chain of simple operations specific to a particular set of parameters. Constants derived from the set of parameters are inlined. Here is a fragment of partial-evaluation output for a combined modular multiplication and carrying operation for $\mathbf{F}_{2^{255}-19}$ with 10 limbs:

```
let fg0 := f0*g0 + 19*(2*f9*g1 + f8*g2 +...)in
let fg1 := fg0>>26 + f1*g0 + f0*g1 + 19*... in
let fg2 := fg1>>25 + f2*g0 + 2*f1*g1 +   ... in
let fg3 := fg2>>26 + f3*g0 +   f2*g1 +   ... in
```

We would like to point out that the same multiplication procedure (minus the bit-shifted carries from one output limb to another) appears in [11, page 12] as a performance-engineering achievement of its own. Our framework generates it and its many variants automatically by partial evaluation, from a multiplication implementation proved correct once and for all.

Conversely, the generic OCaml code extracted in [35] was reported to be over 100 times slower than a reference C implementation, and they pointed out the use of boxed integers and boxed arrays as the main reason for the slowdown. Importantly, this performance-oriented transformation is easily available to us *because* our generic code is purely functional, as opposed to their stateful procedures. Because this transformation proceeds using the built-in rules of computation of Gallina, the specification language of Coq, no proof is required. Just this transformation alone speeds the multiplication code up by a factor of 13.

Still, we had to solve a few engineering challenges, principally informing Coq about which functions to evaluate at compile time and which to save for runtime. We defined aliased versions of several basic arithmetic operations, associating one flavor of each operator with compile-time evaluation, and provided that list of operators to Coq as a whitelist of functions to inline at call sites.

4.2 Bounds Checking

Although the high-level algorithms were designed specifically to avoid overflowing or underflowing machine-integer registers holding the limbs, we cannot rule these out in a generic way without falling back to lower-performance algorithms. Instead, we perform a separate analysis on the partially evaluated code. The requirements for bounds on machine integers are that (a) no primitive machine operation overflows or underflows a (32-/64-bit) word, and (b) the output of any field operation must be at least as tightly bounded as the input to that operation. (Requirement (b) is necessary to allow chaining of field operations.) In practice, finding bounds that satisfy both (a) and (b) is done by educated guessing, and the bounds are dependent on the particular prime being used, the limb widths, and the machine architecture. Because guessing the right bounds is beyond the scope of our automatic synthesis, we take the bounds as part of the input and use a verified, proof-generating checker to ensure that the bounds satisfy these properties. We will use the GHC Haskell compiler in the end, and checking the bounds statically produces code that is 8 times faster than GHC's approach of choosing between machine words and a big-integer implementation at runtime.

Given simple straightline code for each operation and some upper and lower bounds for the inputs, it is straightforward to track the bounds throughout the procedure and establish bounds at the output. We implement bounds analysis after we have already performed all optimizations that make sense in non-assembly code, and performed partial evaluation.

Because Coq is slow at manipulating large terms, and we must analyze hundreds of lines of straight-line code, we do bounds analysis reflectively [15], meaning we define a bounds checker as a formal functional program and prove its soundness once and for all, so that for individual curves we merely run the checker and appeal to its soundness theorem. We encode syntax trees for a subset of Gallina, with simple arithmetic operations and let binders, using parametric higher-order abstract syntax [17], one solution to the surprisingly vexing problem of encoding lambda-term syntax in proof assistants. Bounds analysis proceeds by abstract interpretation on a type of bounded words, which we prove correct in a generic fashion.

We call attention to two features of our reflective automation that the reader may find interesting. First, we decided to flatten let bindings in this stage; this consists, for example, of turning

```
let x := (let y := a + b in y + y) in x + x
```

into

```
let y := a + b in let x := y + y in x + x
```

This transformation is necessary for our eventual goal of synthesizing assembly code. We found that the reflective pipeline that transformed Gallina code operating on unbounded integers into code operating on bounded words was the most convenient place to perform this change.

Second, we found it convenient to represent each function in fully uncurried form, as taking a single tuple argument, and to represent a let binder as binding a number of variables at once. Thus, if `adc` takes three arguments (two values to add and a Boolean carry flag) and returns a pair of values (a Boolean carry flag and the sum of the operands), we would represent binding the return of `adc` to a variable as:

```
let (CF', retv) := adc (a, b, CF) in ...
```

4.3 Double-Word Operations

While the algorithms described in section 3.2 require after-the-fact bounds analysis due to delayed carries, which are useful for performance in many scenarios, that algorithm only works for pseudo-Mersenne primes. When the prime is not pseudo-Mersenne, for example, `p256` $= 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$, we use simpler algorithms for modular reduction that admit a much simpler form of boundedness guarantees.

These algorithms, described in section 4.4, rely only on the ability to do arithmetic on n-bit integers for fixed n (e.g., 256). We built a library to synthesize the operations needed for these algorithms.

Each synthesis rule shows how to construct an operation for n-bit integers from simpler operations on the same size, or any supported operation on $n/2$-bit integers. Therefore, a 256-bit addition is constructed from a 128-bit addition, which is constructed from a 64-bit addition, and so on. Each rule is written down and proven correct once and used several times during synthesis.

4.4 Barrett Reduction and Montgomery Form

The word-size-doubling constructions in section 4.3 give an alternative (sometimes faster, sometimes slower) way of implementing modular reduction without making any assumptions about the value of the prime (only its size). In particular, standard techniques known as Barrett reduction and Montgomery-form reduction allow for reduction modulo any prime at the cost of a handful of simpler arithmetic operations and some prime-specific precomputation. It should be noted that both of these are informal implementation strategies rather than specific formulas; there are multiple variations of each with different performance characteristics. When prototyping our modular-arithmetic implementations, we transcribed and proved correct a total of 5 different sets of parametric formulas. Each formalization reads like (and in one case is based on) a Wikipedia article about the strategy; all 5 add up to 750 lines of code and proof.

For example, reduction modulo `p256` is implemented as using the "REDC" variant of Montgomery reduction, with big-integer arithmetic operations synthesized as appropriate. On the other hand, arithmetic modulo the order of the Curve25519 group, $l \approx 2^{252} + 1.3 \cdot 2^{124}$, is implemented using Barrett-style formulas, mirroring the choice of the fastest public-domain implementations [1, ed25519-amd64-51-30k]. Choosing which formulas to use is not within the scope of our library, and as in the two cases shown, these choices can be made based on existing work with little effort.

5. Case Study: EdDSA

Our main case study is a digital-signature scheme called EdDSA, used for client and server authentication in SSH, TOR, OpenBSD, Signal, a yet-to-be-finalized TLS/x509 certificate type [20], and in numerous less prominent systems [3]. The main goal of this case study is to validate our modular-arithmetic correctness theorems: it would be a pity if the verified specification of the optimized arithmetic code were not complete enough to allow for verified composition. Establishing the correctness of an EdDSA implementation involves a number of interesting proofs that rely on the correctness of modular arithmetic but are otherwise orthogonal. We also included a representative subset of high-level optimizations used in the fastest EdDSA implementations but did not aim for performance parity.

5.1 Fidelity of High-Level Specifications

To the best of our knowledge, our specifications match the descriptions of the relevant cryptographic primitives and mathematical structures more closely than any other specification against which an implementation has been formally verified. As an example, here is our Coq specification of Edwards curves.

```
Definition point := { P : F*F
                    | let '(x,y) := P in
                      a*x^2+y^2 = 1+d*x^2*y^2 }.
Definition coordinates (P:point)
  : (F*F) := proj1_sig P.
Program Definition zero : point := (0, 1).
Program Definition add (P1 P2:point) : point :=
exist _ (
  let (x1, y1) := coordinates P1 in
  let (x2, y2) := coordinates P2 in
    (((x1*y2  +  y1*x2)/(1 + d*x1*x2*y1*y2)),
     ((y1*y2 - a*x1*x2)/(1 - d*x1*x2*y1*y2))))_.
```

The main difference between this and the English-language definition of Edwards curves is an artifact of encoding of subsets in Coq – each point is represented as its coordinates along with a proof of the invariant that the coordinates are on the curve; `proj1_sig` strips this proof, and `exist _ P _` attaches a new one (which is located automatically by Coq's `Program` machinery [32]). Specifications that do not introduce invariants are even simpler; here is our definition of when an EdDSA signature should be accepted:

```
exists A S R, Eenc A = pubkey /\
  Eenc R ++ Senc S = signature /\
  S * B == R+(H(Eenc R++EencA++msg) mod l)*A.
```

5.2 Optimized Point Formats

Cryptography experts have developed faster representations of elliptic-curve points. We use an extended representation of twisted Edwards curves [7], with each point encoded as (X, Y, Z, T), representing the 2-coordinate point $(X/Z, Y/Z)$, with the side conditions $Z \neq 0$ and $T/Z = XY$. Having an explicit field element Z that corresponds to "the denominator" of X and Y enables the addition formulas to be written without any expensive field-element inversions or divisions.

To show that the group formed by the fast 4-coordinate addition formulas is isomorphic to the specification group, we translate the Sage script in [7] to Coq tactics, relying heavily on the `nsatz` [28] tactic, which internally uses Gröbner-basis computation for finding proof certificates. The same strategy (along with some simple facts about when a product can be nonzero) applies to proving that Edwards-curve points indeed form a group or that the Edwards addition-law denominator is never zero (theorem 3.3 in [9]).

5.3 Space-Efficient Representations

Edwards curve points (x, y) in EdDSA are transmitted as $(y, x \bmod 2)$ to save space, and the recipient is expected to solve the curve equation for x^2 and then pick the unique value for x such that the parity is correct (x and $-x$ have different parity if q is odd). Our verification covers both encoding a point in precisely the specified way and solving for the omitted coordinate given the sign bit, and we are not aware of any previous verification of these procedures – both are conceptually simple but require simultaneous attention to nontrivial number-theoretic properties and the exact wire format.

Furthermore, while signature verification would naively involve decoding two points (signature nonce R and public key A), we include (and verify) a well-known optimization that involves decoding A, solving for R, encoding it, and comparing the encodings – again, saving a field-element inversion and square-root computation.

5.4 Powers, Inverses, Square Roots

From the basic field operations, we can construct a more complete set of necessary operations on field elements, relying on a suite of nonobvious implementation techniques.

For **exponentiation** by a constant, we go beyond the *binary exponentiation* that would, for instance, express x^{15} with 6 multiplications as $x^{15} = x \times (x \times (x \times x^2)^2)^2$. Instead, we parameterize exponentiation over arbitrary *addition chains* that encode the sharing of subcomputations, so that we can compute x^{15} with only 5 multiplications as $x^{15} = x^3 \times ((x^3)^2)^2$.

For **inversion** (finding a multiplicative inverse), we use Fermat's Little Theorem (whose proof we import from the Coqprime [33] library) to show that a^{p-2} is the multiplicative inverse of a modulo p, so that our exponentiation routine provides the main ingredient.

For finding **square roots** (modulo prime p), we employ various prime-specific tricks for fast execution based on Euler's criterion. For instance, when $p \equiv 3 \mod 4$, the square root of a (if a has a square root) is given by $a^{\frac{p+1}{4}}$, again relying on exponentiation.

6. Evaluation

We will evaluate our implementation using the three criteria described in the introduction: rigor (section 6.1), generality (section 6.3), and effort (section 6.3). We will also compare our performance to real-world implementations (section 6.2).

6.1 Safety From Bugs

We analyzed deployed cryptographic software for a sample of functional-correctness bugs specific to the functions being implemented (to exclude memory-management errors or timing side channels). Our sample includes the first 15 bugs we found fitting the criterion of specificity to the implemented function; we did not intentionally exclude bugs for any other reason, although a more thorough search would probably uncover more bugs. We observed that while the mistakes were often "small" in the sense that the difference between the original and corrected versions was minimal (in one case, a single character), understanding why one is correct and the other is not requires significant contextual information, sometimes across multiple abstraction layers. In sharp contrast with the top-level specification, it is nontrivial to write the specification of a subroutine that captures all required behaviors and yet allows for important optimizations.

6.1.1 Multiword Arithmetic Bugs

Of the 15 bugs, 11 had to do with low-level multiword arithmetic. These are summarized in Table 1. In this category, it is nontrivial to distinguish between design errors (code correctly implements the programmer's flawed understanding) and coding mistakes (typos, missed low-level details) based on the code itself, so we refer to relevant bug-tracker discussion if available. Similarly, it is extremely difficult to estimate the security impact of these bugs: [4] shows a sophisticated exploit against OpenSSL bug 1593, and Bernstein and Schwabe estimate [10] that a well-equipped attacker would be able to exploit OpenSSL CVE-2015-3193, but we do not know about all of the bugs. A detailed analysis of exploitability is outside the scope of this project, and we choose not to speculate.

Here are three example bugs for which an explanation was available:

- The TweetNaCl paper [14] describes a typo of unknown impact in `ed25519-amd64-64-24k`: `r1 += 0 + carry` should have been `r2 += 0 + carry` instead. Authors noted that this line was one of 16,184 similar lines, and the issue would not have been caught by random tests.
- OpenSSL bug 1593 was traced back to confusion between the postconditions of exact division with remainder and an operation like our *split* that produces a q and r s.t. $x = qm + r$, but does not guarantee that r is the smallest possible. The probability of a random test triggering this bug was bounded to $10 \cdot 2^{-29}$.
- One of the two bugs uncovered in OpenSSL issue 3607 was summarized by its author as "`Got math wrong :-(`". The discussion was concluded when the patched version was found to be "good for ~6B random tests" and the reviewer saw that "there aren't any low-hanging bugs left."

6.1.2 Higher-Level Bugs

While field arithmetic accounts for the nastiest bugs we have seen, the following are good examples of how increased rigor would have helped higher-level code: Jager et al. demonstrate [19] how, because an elliptic-curve point-decoding function failed to establish that the point indeed lies on the elliptic curve, devastating remote attacks were enabled against TLS/HTTPS security implemented by Oracle Java and Bouncy Castle libraries. Google End-to-End issue 340 describes a conceptually similar confusion about the members of an elliptic curve with no known adverse consequences. CVE-2006-4339 involves improper validation during parsing that allowed for bogus RSA signatures to pass as valid. Even more embarrassingly, Socat security advisory 7 admits unbounded loss of security due to a hardcoded constant that was required to be prime actually being composite (and of unknown origin).

Table 1: Multiword Arithmetic Bugs

Software	Reference	Summary
OpenSSL	CVE-2015-3193	Integer overflow (next to comment "can this overflow?")
OpenSSL	commit 0c687d7e	Possible overflow (failed manual analysis)
OpenSSL	issue 3607	Integer overflow
OpenSSL	issue 3607	Precondition/postcondition confusion
OpenSSL	issue 1593	Precondition/postcondition confusion
`curve-donna-c32`	commit 8edc799f	Incomplete modular reduction
Golang `math/big`	issue 13515	Incorrect carrying
OpenSSL	CVE-2016-7055	Incorrect carrying
Nettle	commit 09e3ce4d	Incorrect carrying
`ed25519-amd64-64-24k`	TweetNaCl (p. 2)	Typo among 16,184 similar lines
OpenSSL	CVE-2014-3570	Incorrect results when squaring field elements

6.1.3 Our Contribution

Our verification rules out all functional-correctness bugs (historic and hypothetical) that we are aware of, including those described above. Because we use bounded types, results that are incorrect due to integer overflow are covered (see section 4.2). Our high-level specifications ensure that our operations compose correctly; in order to do so, they must produce correct output in all cases. We represent elliptic-curve points with a dependent type that not only includes coordinates but also requires the coordinates to be on the curve; it is impossible to parse a point that is not in fact on the curve, and we avoid the problems encountered by Oracle Java and Bouncy Castle.

Furthermore, our verification has already uncovered subtle issues with real-world implementations on two occasions. First, in the process of producing our verified implementations, we discovered a troubling difference between what could be verified to be correct and similar implementations deployed by a well-known technology company. The implementation we studied performed Barrett-style reduction modulo two different primes and was justified by the same correctness argument for both of them. The code featured a novel optimization to perform one conditional subtraction of the modulus instead of two. The generic justification did not pass verification in Coq, was found to be faulty, and has not found a fix to this day. For one of the two moduli, we were able to check the computation by enumerating all semantically different inputs, but doing the same for the other one would have been computationally infeasible. After significant investigation failed to turn up any reason why the remaining code would be correct, the company removed the new optimization from production.

Second, we also discovered a discrepancy between the paper specification (and Python reference implementation) of ed25519 and all the most optimized implementations we tested (C "ref", ed25519-donna, ed25519-amd64-51-30k). In particular, a malicious actor would be able to create a signature that is considered good by any implementation that follows the specification to the letter but considered invalid by the optimized implementation. The difference comes from the fact that the point $(h \bmod l)P$ is equal to hP only if P has order l, but Curve25519 also contains points of other orders, even though a good signer never generates them. The optimization is sound in the sense that it makes no new signatures valid, and it significantly simplifies implementation, so we argue that the specification should be changed to allow for that optimization. The latest IETF draft [20], due to an independent fix, contains this updated specification.

6.2 Performance

To measure the performance of our synthesized code, we use Coq's extraction mechanism to translate the output to Haskell. We have the extraction map key types in Coq to their Haskell counterparts (in particular, `word 64` becomes `Data.Word.Word64`) and instruct GHC to use strict evaluation and inline field-arithmetic functions.

The performance of our synthesized code reflects the optimizations we verified and integrated. In particular, our synthesis output reliably outperforms generic code and naive specialized implementations, comes close to similarly optimized handwritten code, and lags behind the world-champion implementations that incorporate additional optimizations. Tables 2 and 3 contain our own benchmark results on a 2.6GHz Intel Broadwell i7-5600U processor along with SUPERCOP results [1] from machine `skylake` scaled to the former's clock frequency.

The performance ranking of our synthesized code on the X25519 key-agreement benchmark is limited by our suboptimal choice of limb widths and complete lack of instruction-level optimization: an implementation written in C with the same limb widths is 40% faster than

Table 2: X25519 Diffie-Hellman Handshake

implementation	time	limbwidths	lang.
`amd64-51`	69μs	51	qhasm
`donna-c64`	71μs	51	C
`donna-2526`	237μs	25.5	C
ours	384μs	25.5	Haskell
GMPtoy	1022μs	-	C/Haskell
`ref`	2351μs	8	C
ECC-star [35]	20000μs	51	OCaml

ours, and using 51-bit limbs saves another factor of 4 on our field multiplication. We are working on extending our synthesis process to support generating code for 51-bit limbs: all layers that work with mathematical integers already support it, but the verified translation from arbitrary-precision integers to 64-bit words, described in section 4.3, would need to detect and separately handle 128-bit products of 64-bit integers that need to be split across two variables. We also included for reference a naive non-constant-time (and thus insecure) implementation where all field arithmetic is performed by the GNU Multiple Precision arithmetic library.

Table 3: Ed25519 signing

implementation	time	exp.	limbs	lang.
`amd64-51-30k`	19μs	hex	51	qhasm
`donna-c64`	22μs	hex	51	C
`donna-2526-64`	44μs	hex	25.5	C
`ref`	443μs	hex	8	C
ours	1529μs	bin	25.5	Haskell
`ed25519.py`	2010000μs	bin	-	Python
our spec	$\sim 2^{250}\mu$s	n/a	-	Coq

While X25519 primarily exercises the lower layers of our framework (finite-field operations, efficient representation of large bounded integers), the performance of an Ed25519 implementation depends even more heavily on the handling of elliptic-curve points and in particular the implementation of point-scalar multiplication. As seen in table 3, an "unoptimized" implementation derived directly from the specification would be completely intractable to execute, and replacing point multiplication by repeated addition with an analogue of binary exponentiation (as in the original reference implementation `ed25519.py`) still leaves several orders of magnitude of speed to be gained. Our implementation further improves on that by using inversion-free point-addition formulas in extended coordinates, coming to within 4x of the C "ref" implementation that combines the best-known elliptic-curve optimizations with a naive field-arithmetic library. The fastest implementations use precomputed tables, represent points differently depending on what operations are performed on that point, and use "hexadecimal exponentiation" (handling a fixed window of 4 exponent bits at a time).

Finally, we are excited to report that the implementation of Barrett reduction discussed in the end of section 6.1.3, which we synthesized to match third-party code, actually ended up 10% faster than the hand-optimized target. Our synthesis pipeline correctly identified and eliminated a redundant computation that the programmer missed.

6.3 Effort

The framework described in this paper was designed and implemented in less than two person-years of work. This is on par with the timescales of cryptographic implementation development: for example, the project that produced the qhasm implementations of X25519 and ed25519 (discussed earlier section 6.2) took 38 months [22], and the ed25519 paper [12] lists 5 authors. Of course, the original ed25519 developers had to come up with the optimizations themselves, which is at least as difficult as our coming up with a strategy for verifying them.

We do not have a direct indication of the effort to later verify the body of the innermost loop of the X25519 implementation as done by Chen et al. [16], but we expect their smaller scope and looser integration story between different formal-methods tools simplified implementation, which involved 8 authors. The paper also indicates a 27-fold increase in proof steps between the fastest multiplication modulo $2^{127} - 1$ and the fastest multiplication modulo $2^{255} - 19$. The authors of gfverif [10] judge this previous verification to have required "much more effort" than checking similar C code with gfverif, but they also emphasize that the latter "should not be relied upon." The verification of a new, much slower implementation of arithmetic in the same field by Zinzindohoue al. [35] roughly matches ours in terms of proof size (5800 lines), but the later synthesis steps that are key to our better performance require additional proof, adding up to a total of 8000 lines. They still require a significant overhead of 600 handwritten lines per curve (compared to our 1 line), and none of the three projects provide comprehensive whiteboard-level specifications.

Where synthesis really pays off is producing new code using a known strategy. For example, when we read the work of Zinzindohoue et al. [35] and decided to synthesize a X25519 implementation to compare performance, it took one person less than 8 hours to synthesize code using the optimizations in our library, prove the correctness of the composition, and extract and benchmark Haskell code. In the "real world," even smaller changes like replacing the prime modulus require from-scratch reimplementation. These delays have unfortunate security consequences: for example, the EdDSA RFC [20] recommends Ed25519 over Ed448 in cases where the inferior security margin is tolerable and notes that the latter has “much worse support” on the implementation front. In our framework, creating an implementation with a single parameter takes less than 10 minutes: one has to copy the file that specifies the synthesis parameters and change what is desired. We practiced this exercise on 5 pseudo-Mersenne primes from the SafeCurves list of primes recommended for use in elliptic-curve cryptography [8]. In particular, we generated code for $2^{221} - 3$, $2^{251} - 9$, $2^{255} - 19$, $2^{414} - 17$, and $2^{521} - 1$. Nearly all of the process was automatic; the only input necessary to generate verified, optimized field arithmetic for a new prime was a tiny JSON file. This is the input for the prime $2^{221} - 3$:

```
{ "k" : 221, "c" : 3, "n" : 8, "w" : 32,
  "ch" : "[0;1;2;3;4;5;6;7;0;1]" }
```

The parameters n and w correspond to number of limbs and machine-word size (though due to current lack of support for multiple word sizes discussed in Future Work, this code would actually be generated for 64-bit). The *ch* parameter is the carry chain used after multiplication. Determining the optimal carry chain is currently not a generically solved problem, so it must be specified by the programmer. (However, our machinery will check for integer overflow automatically should the programmer fail to carry enough.) A short Python script does text replacement on a template, producing Coq source that differs only superficially, despite the fact that the final generated code will include operations in entirely different base systems.

7. Related Work

As discussed in the introduction and the evaluation section, prior projects in verifying field-specific modular-arithmetic code have had significantly larger trusted bases and less concise specifications than ours, and are either less much more work-intensive per implementation [16] or result in significantly slower code [35].

Performance-oriented synthesis of domain-specific code (without proofs of correctness) has previously been done using explicit templates (e.g. Template Haskell [31]) and more sophisticated multistage programming (e.g. Lightweight Modular Staging [30]). More specialized frameworks along these lines include FFTW [18] and Spiral [26]. Out of these, our synthesis strategy is most similar to LMS, differing mainly in the choice of using existing (proof-generating) Coq facilities for controlled partial evaluation and rewriting rather than implementing them ourselves.

Myreen and Curello verified a general-purpose big-integer library [24]. The code uses a hardcoded uniform base system, does not include specialized modular-reduction optimizations, and does not run in constant time. However, their verification extends all the way down to AMD64 assembly using verified decompilation. The proof effort is roughly similar to ours (6227 lines of HOL).

While verified compilers (e.g., CakeML [21], CompCert [23]) and translation validators [25] are useful for creating soundly optimized versions of a reference program, we are not aware of any that could cope with abstraction-level-collapsing synthesis as done in this work or LMS.

Verification of cryptographic protocols (e.g., CertiCrypt [34], FCF [27]) is complementary to this work: given a good formal specification of a protocol, it can be shown separately that an implementation corresponds to the protocol (our contribution for EdDSA) and that the protocol is secure (out of scope for this paper). The work by Beringer et al. [5] is a good example of this pattern, composing a protocol-security proof, a correctness proof for its C-level implementation, and a correctness proof for the C compiler.

8. Future Work

There are several additional blockers to the use of our synthesized implementations in any real product. Most importantly, to be linked against programs in languages other than Haskell, our synthesized code would need to be extracted to a low-level language with no managed runtime. Mechanically, this is perfectly tractable right now: a purely syntactic change of `let x := Word64.add a b in ...` to `uint64_t x = a + b;` would yield valid C code, but ideally the transformation would be accompanied by a proof of correctness (removing GHC and Coq’s extraction machinery from our trusted base).

We are working on a synthesis pass to choose fixed representations of different sizes of integers. This would enable us to use base-2^{51} and thus bridge most of the X25519 performance gap. The remaining 40% is due to instruction-level optimization in other implementations. Similarly, while our current EdDSA implementation is foundationally verified and fast enough for some applications, the remaining elliptic-curve-level optimizations would need to be included to achieve competitive performance (a prerequisite for large-scale adoption). At the very end, the output machine code should be checked against the semantics of the target machine to make sure that there is no data flow from secret inputs to execution time (e.g. through branch conditions or array indices) – while our code is written with constant time in mind, GHC (or any other sophisticated compiler we are aware of) is not designed to preserve that property.

In addition to improving the current use cases of this library, we are optimistic that the same approach will be useful in other cryptographic domains: most obviously, hyper-elliptic-curve cryptography, but also lattice-based cryptography and possibly also RSA with a fixed modulus size.

A. Elliptic Curves and Cryptography

Elliptic curves are defined over finite fields $\mathbf{F}_q$ (integers modulo q) as sets of points that satisfy some predicate (the curve equation). We turn curves into groups by adding an addition law. A twisted Edwards curve with parameters a and d is defined as follows:

$$\{(x, y) \in \mathbf{F} \times \mathbf{F} \mid ax^2 + y^2 = 1 + dx^2y^2\}$$

$$(x_1, y_1) + (x_2, y_2) = \left(\frac{x_1y_2 + x_2y_1}{1 + dx_1y_1x_2y_2}, \frac{y_1y_2 - ax_1x_2}{1 - dx_1y_1x_2y_2}\right)$$

Multiplication of a point by an integer is defined as repeated addition, and $(0, 1)$ is the identity element. There are other families of elliptic curves (more and less general), and the "same" curve (up to isomorphism) can be represented in different coordinate systems. In cryptography, the coordinate system and parameters are chosen for best security and performance if at all possible, and a and d are chosen such that the denominator is never zero. Our chosen example curve is Curve25519 [6], because it is highly optimizable and is used in a broad swath of applications [2, 3]. Curve25519 uses $q = 2^{255} - 19$, $a = -1$, and $d = -121665/121666$. Importantly, the curve is specified in the form best for comprehension and security analysis, but implementations can use any coordinate system for which there exists an efficiently computable isomorphism.

A.1 EdDSA Digital Signatures

A motivating application of ECC is the EdDSA signature scheme, which one host can use to attach a digital signature to a network message. The signature is relatively efficient to compute, given knowledge of a secret key, but intractable to forge without the key. The general EdDSA signature scheme with 11 parameters is defined rigorously in [13], which is the basis of our formalization. The definitions in [12, 20] are special cases of the former, but we follow the last one here for simplicity. Parameters include:

- An Edwards curve
- A canonical encoding of points
- "Base" point B
- Cryptographic hash function H
- An integer l s.t. $lB = (0, 1)$

For a secret integer a, the public key is $A = aB$. Then the signature on a message M is defined as the pair $(rB, (r + H(rB, A, M)a) \bmod l)$, where r is a pseudorandom value derived from a secret and M using H. To verify an alleged signature (R, S) by A on message M, it is sufficient to check $SB = R + H(R, A, M)A$. Some standards mandate different checks, but these are not necessary for detecting forgeries. It is critical that the signer does not reveal any information about r for any of its signatures – otherwise the signing equation could be solved for a.

A.2 Diffie-Hellman Key Exchange

Given a common base point B, two parties (with secret integers x and y and public-key points $X = xB$ and $Y = yB$) who are able to mutually authenticate each other can agree on a shared secret by publishing X and Y and then computing

$$xY = x(yB) = xyB = yxB = y(xB) = yX$$

It is permissible to perform the key exchange first and then authenticate the choices of X, Y before continuing, but it is not okay to use xyB as the basis of that authentication or to authenticate xyB instead. It is safe to reuse x and X for multiple key agreements, but erasing x can avoid compromise of finished sessions in case of attack.

A.3 TLS 1.3

The work-in-progress version of the Transport Layer Security standard allows connection establishment using the two described algorithms as of the latest IETF drafts [20, 29]. In particular, each connection begins with both the client and the server transmitting their chosen signature and key-agreement mechanisms, the information necessary for key agreement (here X), and a signature on the hash of all session-initiation messages (including the ones described here). Elliptic-curve cryptography is used only for connection initiation and authentication; subsequent application messages are encrypted and authenticated using simpler methods that rely on the presence of a shared key, for example `chacha20poly1305`.

B. Techniques

In the course of this project, we came up with a handful of useful techniques and patterns to simplify our verification task. Of these, a few address generic problems that future researchers might face; we summarize those here.

B.1 Representation Proofs

Multiple optimizations verified in our library rely on changing representation of the values that are being computed on throughout the entire computation. In particular, given an existing reference implementation of some implementation of an abstract type T_{REF} with operations $f : T_{\text{REF}} \to T_{\text{REF}}$, we wish to show that another type T_{OPT} with corresponding operations behaves equivalently in the sense required by the Coq rewriting mechanisms we use to synthesize code. In particular, the following requirements need to be met to push a conversion from T_{REF} to T_{OPT} towards the leaves of an expression tree:

1. $\mathrm{OPT}(f(x)) = f_{\mathrm{OPT}}(\mathrm{OPT}(x))$
2. f_{OPT} is proper: if $\mathrm{REF}(x_{\mathrm{OPT}}) = \mathrm{REF}(y_{\mathrm{OPT}})$ then $\mathrm{REF}(f_{\mathrm{OPT}}(x_{\mathrm{OPT}})) = \mathrm{REF}(f_{\mathrm{OPT}}(y_{\mathrm{OPT}}))$

If the equivalence of T_{OPT} values corresponds to equality of their REF images, this can be pictorially represented as the following commuting diagram:

$$\begin{tikzcd}
T_{\mathrm{REF}} \arrow[r, "f"] \arrow[d, "\mathrm{OPT}"'] & T_{\mathrm{REF}} \arrow[d, "\mathrm{OPT}"] \\
T_{\mathrm{OPT}} \arrow[r, "f_{\mathrm{OPT}}"'] & T_{\mathrm{OPT}}
\end{tikzcd}$$

$$\begin{tikzcd}
{} \arrow[r, Rightarrow, "\text{Proper}"'] & {}
\end{tikzcd}$$

However, proving both properties directly would result in a duplication of work for each function f. The following "flipped" formulation, which can be easily certified to imply the former, requires only one proof about each function, and is in our experience much easier to prove:

$$\begin{tikzcd}
T_{\mathrm{REF}} \arrow[r, "f"] \arrow[d, bend right, "\mathrm{OPT}"'] & T_{\mathrm{REF}} \\
T_{\mathrm{OPT}} \arrow[r, "f_{\mathrm{OPT}}"'] \arrow[u, bend right, "\mathrm{REF}"'] & T_{\mathrm{OPT}} \arrow[u, "\mathrm{REF}"']
\end{tikzcd}$$

1. $\mathrm{REF}(f_{\mathrm{OPT}}(x_{\mathrm{OPT}})) = f_{\mathrm{REF}}(x_{\mathrm{OPT}})$
2. faithful representation: $\mathrm{REF}(\mathrm{OPT}(x)) = x$

B.2 Selective Partial Evaluation

In section 4.1, we discussed partially evaluating code such that function calls were inlined and computed as far as possible, leaving only low-level operations (in particular, operations with corresponding assembly instructions). We did this by "whitelisting" those functions, telling Coq to inline and compute everything else. But in some places, we did actually want to inline those functions. For example, we would not want to do compile-time computation of an addition that operated on two pieces of input, but we do want compile-time computation of additions that operate on variables that are known at compile time but are unknown to generic functions. In order to differentiate these cases, we needed a way to "mark" the instances of addition, bitshifting, etc. that we *did* want Coq to compute. In order to do this, we defined new versions of the operations that were equivalent to the old ones but were computed as far as Coq could compute them with no input. Then we replaced the instances we wanted to compute with these new operations, and they were not preserved by the whitelist.

B.3 Optimization through Interactive Proofs

Many steps in our process, including the refinements of field operations described in section 4.1 and the marking of functions in section B.2, required us to synthesize new equivalent versions of functions after more variables were known. One way to do this is to write a new function and then prove it equivalent to the old one; however, even with good proof automation, this is a fairly manual process. A change done midway through the pipeline does not automatically propagate; it needs to be manually changed in every later step. So we used a more automated technique. It is probably not novel; however, it is worth noting as a useful and little-discussed method.

1. To start interactive optimization, state the goal as an informative existential: `Definition f_opt_sig : { f_opt | forall x, f_opt x = f x }.`
2. Start proving the correctness while leaving `f_opt` to be determined in process by running `eexists`.
3. Perform any `rewrite` and other desired simplification on the right-hand-side of the equality.
4. Run `reflexivity` to unify the two sides, implicitly determining `f_opt`.
5. Use `Definition f_opt := Eval cbv [proj1_sig f_opt_sig] in f_opt_sig` to retrieve the function.

References

[1] ebacs: Ecrypt benchmarking of cryptographic systems. URL https://bench.cr.yp.to/supercop/supercop-20161026.tar.xz.

[2] Things that use curve25519, . URL https://ianix.com/pub/curve25519-deployment.html.

[3] Things that use ed25519, . URL https://ianix.com/pub/ed25519-deployment.html.

[4] Practical realisation and elimination of an ECC-related software bug attack. 2011. URL https://eprint.iacr.org/2011/633.pdf.

[5] L. Beringer, A. Petcher, K. Q. Ye, and A. W. Appel. Verified correctness and security of OpenSSL HMAC. In *24th USENIX Security Symposium*, pages 207–221, Aug. 2015.

[6] D. J. Bernstein. Curve25519: new Diffie-Hellman speed records. In *Public Key Cryptography - PKC 2006. Proceedings of the 9th International Conference on Theory and Practice in Public-Key Cryptography, New York, NY, USA, April 24-26*. Springer-Verlag. URL http://cr.yp.to/papers.html#curve25519.

[7] D. J. Bernstein and T. Lange. Explicit-formulas database: Extended coordinates for twisted edwards curves, . URL https://hyperelliptic.org/EFD/g1p/auto-twisted-extended.html.

[8] D. J. Bernstein and T. Lange. Safecurves: choosing safe curves for elliptic-curve cryptography, . URL http://safecurves.cr.yp.to.

[9] D. J. Bernstein and T. Lange. Faster addition and doubling on elliptic curves. In *Proceedings of the Advances in Crypotology 13th International Conference on Theory and Application of Cryptology and Information Security*, ASIACRYPT'07, pages 29–50, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 3-540-76899-8, 978-3-540-76899-9. URL https://eprint.iacr.org/2007/286.

[10] D. J. Bernstein and P. Schwabe. URL http://gfverif.cryptojedi.org/.

[11] D. J. Bernstein and P. Schwabe. NEON crypto. In E. Prouff and P. Schaumont, editors, *Cryptographic Hardware and Embedded Systems – CHES 2012*, volume 7428 of *Lecture Notes in Computer Science*, pages 320–339. Springer-Verlag Berlin Heidelberg, 2012. URL http://cryptojedi.org/papers/#neoncrypto. Document ID: 9b53e3cd38944dcc8baf4753eeb1c5e7.

[12] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang. High-speed high-security signatures. *Journal of Cryptographic Engineering*, 2(2):77–89, 2012. URL http://cryptojedi.org/papers/#ed25519. Document-ID: a1a62a2f76d23f65d622484ddd09caf8.

[13] D. J. Bernstein, S. Josefsson, T. Lange, P. Schwabe, and B.-Y. Yang. EdDSA for more curves, 2015. URL http://cryptojedi.org/papers/#eddsa.

[14] D. J. Bernstein, B. van Gastel, W. Janssen, T. Lange, P. Schwabe, and S. Smetsers. TweetNaCl: A crypto library in 100 tweets. In D. Aranha and A. Menezes, editors, *Progress in Cryptology – LATINCRYPT 2014*, volume 8895 of *Lecture Notes in Computer Science*, pages 64–83. Springer-Verlag Berlin Heidelberg, 2015. Document ID: c74b5bbf605ba02ad8d9e49f04aca9a2, http://cryptojedi.org/papers/#tweetnacl.

[15] S. Boutin. Using reflection to build efficient and certified decision procedures. In *Proc. TACS*, 1997.

[16] Y.-F. Chen, C.-H. Hsu, H.-H. Lin, P. Schwabe, M.-H. Tsai, B.-Y. Wang, B.-Y. Yang, and S.-Y. Yang. Verifying Curve25519 software. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, CCS'14*, pages 299–309. ACM, 2014. URL http://cryptojedi.org/papers/#verify25519. Document ID: 55ab8668ce87d857c02a5b2d56d7da38.

[17] A. Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *ICFP'08: Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, September 2008. URL http://adam.chlipala.net/papers/PhoasICFP08/.

[18] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2): 216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[19] T. Jager, J. Schwenk, and J. Somorovsky. Practical invalid curve attacks on TLS-ECDH. 2015. URL http://euklid.org/pdf/ECC_Invalid_Curve.pdf.

[20] S. Josefsson and I. Liusvaara. Edwards-curve digital signature algorithm (EdDSA). Internet-Draft draft-irtf-cfrg-eddsa-08, IETF Secretariat, August 2016. URL http://www.ietf.org/internet-drafts/draft-irtf-cfrg-eddsa-08.txt.

[21] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens. CakeML: A verified implementation of ML. In *POPL '14: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 179–191. ACM Press, Jan. 2014. doi: 10.1145/2535838.2535841.

[22] T. Lange, D. J. Bernstein, and P. Schwabe. Improved networking and cryptography library. Technical report. URL https://cryptojedi.org/papers/caced25-20110211.pdf.

[23] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[24] M. O. Myreen and G. Curello. A verified bignum implementation in x86-64 machine code. URL http://www.cse.chalmers.se/~myreen/cpp13.pdf.

[25] G. C. Necula. Translation validation for an optimizing compiler. pages 83–94.

[26] M. Püschel, B. Singer, J. Xiong, J. M. F. Moura, J. Johnson, D. Padua, M. M. Veloso, and R. W. Johnson. Spiral: A generator for platform-adapted libraries of signal processing algorithms. *Journal of High Performance Computing and Applications, special issue on Automatic Performance Tuning*, 18(1):21–45, 2004.

[27] A. Petcher and G. Morrisett. The Foundational Cryptography Framework. 2014. URL http://adam.petcher.net/papers/FCF.pdf.

[28] L. Pottier. Connecting Gröbner bases programs with Coq to do proofs in algebra, geometry and arithmetics. *CoRR*, abs/1007.3615, 2010. URL `http://arxiv.org/abs/1007.3615`.

[29] E. Rescorla. The Transport Layer Security (TLS) protocol version 1.3. Internet-Draft draft-ietf-tls-tls13-11, IETF Secretariat, December 2015. URL `http://www.ietf.org/internet-drafts/draft-ietf-tls-tls13-11.txt`. `http://www.ietf.org/internet-drafts/draft-ietf-tls-tls13-11.txt`.

[30] T. Rompf and M. Odersky. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled DSLs. *Commun. ACM*, 55(6), 2012.

[31] T. Sheard and S. P. Jones. Template meta-programming for Haskell. 2 2016. URL `https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/meta-haskell.pdf`. orig. 2002.

[32] M. Sozeau. Subset coercions in Coq. 2006. URL `https://www.irif.fr/~sozeau/research/publications/Subset_Coercions_in_Coq.pdf`.

[33] L. Théry and B. Grégoire. Coqprime. URL `http://coqprime.gforge.inria.fr/`.

[34] S. Zanella-Béguelin. Formal certification of game-based cryptographic proofs. URL `http://software.imdea.org/~szanella/Zanella.2010.PhD.pdf`.

[35] J. K. Zinzindohoue, E.-I. Bartzia, and K. Bhargavan. A verified extensible library of elliptic curves. In *IEEE Computer Security Foundations Symposium (CSF)*, 2016.