Systematic Generation of Fast Elliptic Curve Cryptography Implementations

Andres Erbsen
MIT
Cambridge, MA, USA
andreser@mit.edu

Jade Philipoom
MIT
Cambridge, MA, USA
jadep@mit.edu

Jason Gross
MIT
Cambridge, MA, USA
jgross@mit.edu

Robert Sloan
MIT
Cambridge, MA, USA
varomodt@gmail.com

Adam Chlipala
MIT
Cambridge, MA, USA
adamc@csail.mit.edu

Abstract

Widely used implementations of cryptographic primitives employ number-theoretic optimizations specific to large prime numbers used as moduli of arithmetic. These optimizations have been applied manually by a handful of experts, using informal rules of thumb. We present the first automatic compiler that applies these optimizations, starting from straightforward modular-arithmetic-based algorithms and producing code around 5X faster than with off-the-shelf arbitrary-precision integer libraries for C. Furthermore, our compiler is implemented in the Coq proof assistant; it produces not just C-level code but also proofs of functional correctness. We evaluate the compiler on several key primitives from elliptic curve cryptography.

1 Introduction

Software development today benefits from division of labor. For instance, novices can quickly assemble functional Web applications by delegating most work to featureful open-source frameworks. Experts, too, benefit from reusing complex components, especially when these same people are not also experts on computer performance engineering. A scientist might produce a simulation program, relying critically on a library of optimized data structures and on an optimizing compiler for a high-level language. In well-developed ecosystems of this kind, subject-matter experts can iterate rapidly through the design spaces meaningful to them.

One domain lacking that kind of tooling today is cryptography. The field is exploding, with ongoing experimentation in domains like secure outsourced and multiparty computation. New protocols are being proposed frequently. However, experiments with deploying these protocols are hindered by a reality that most software developers are not aware of: even a competently written C implementation of a new cryptographic primitive will often be 5X slower or worse than what implementation experts know how to build. It is rare for a single person to have the expertise both in protocol/primitive design and in their efficient implementation on commodity processors. Even for that rare person, it is common, in the course of implementing optimizations, to introduce bugs with serious security implications.

Even a 2X performance cost is prohibitive for, e.g., the big Internet companies, operating massive data centers where a cryptographic primitive may be activated millions of times per second. For instance, *elliptic curve cryptography (ECC)* is used preferentially on every new HTTPS connection, with the draft TLS 1.3 protocol that should become the industry standard in the next few years. Companies have enormous incentives to optimize these building blocks. Today's labor cost of manual optimization may be so high that potential users of novel cryptographic functionality never bother to develop related systems.

In this paper, we present *the first automatic compiler performing the number-theoretic optimizations required for competitive elliptic-curve code*, and furthermore, our compiler is implemented in the Coq proof assistant, giving first-principles proofs of correctness, relating generated low-level code to whiteboard-level number theory. For the first time, cryptographic protocol experts have a push-button way to generate fast implementations of new curve variants.

Our generated code does not yet match the performance of world-champion implementations for all curves, but it is a significant advance over what can be implemented without domain-specific optimization. For Curve25519, the one most favored by cryptographers today, we are about 20% off from the latency of the best assembly code. Further advances should be achievable using problem-specific instruction scheduling and register allocation, which we leave for future work. It is conceivable that such work could lead to a fully automatic, correct-by-construction pipeline that produces world-champion assembly implementations from descriptions of elliptic curves.

Our results are already good enough that Google Chrome has adopted our compiler, through the BoringSSL library,

Conference'17, July 2017, Washington, DC, USA
2018. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

replacing previous handwritten C code for Curve25519, incurring performance overhead small enough to be within measurement error. As a consequence, within a year or so, we expect that a significant percentage of all Web client connections will be running our autogenerated, proved-correct code, without the old worries about implementation errors voiding security guarantees.

Which dimensions of variation show up in this domain? The most important one is changing the large prime numbers used as moduli for arithmetic. Number-theoretic optimizations are used to generate code in ways very sensitive to details of the prime numbers. We codify these optimizations, which crypto-implementation experts apply intuitively, in a compiler for the first time.

The situation is also complicated by competing demands of performance and security/privacy. Many of today's most widely used cryptographic primitives can be defined in single pages of pseudocode, and, handed such a piece of paper, the average developer would have little trouble coding up a script using, for instance, Python's arbitrary-precision integers. However, this script would likely use non-constant-time arithmetic operations, leaving it vulnerable to timing attacks, and would have very uncompetitive performance.

The custom code that the experts write often has serious correctness and security bugs. We performed an in-depth analysis of issues from public bug trackers in this domain, with results reported in Appendix A (anonymous supplement). The most common source of defects is the use and implementation of custom representations that split integers into multiple digits of carefully chosen sizes, a subject that will be our main interest in this paper. Our new compiler avoids all of these bugs by construction. It is featureful enough to generate the elliptic-curve implementations used in the TLS protocols. There, every new HTTPS connection must perform *key agreement*, whereby public-key crypto is used to agree on a shared secret, which then drives faster symmetric-key algorithms; and *signature checking*, whereby server certificates are verified for authenticity. Elliptic curves are the mechanism for these tasks most favored by cryptographers today, and TLS 1.3 supports multiple curves, including Curve25519 and NISTP256.

This general area is a fertile one, with many recent projects proving functional correctness and security of crypto-primitive code that has already been written: HACL* [22] for a library in the F* programming language, Jasmin [1] for routines in a cross-platform assembly language, and Vale [7] for metaprograms that generate assembly. Vale's case-study programs mimic standard practice in libraries like OpenSSL, where metaprogramming is used to unroll loops and realize other modest effort savings over writing assembly code directly. However, in all cases mentioned here (and in mainstream libraries), all *curve-specific* aspects of code are handwritten

Input:

```
modulus = 2^256 - 2^224 + 2^192 + 2^96 - 1
architecture = amd64
```

Output:

```
multiply(uint64_t x8, uint64_t x9, uint64_t x7,
         uint64_t x5, uint64_t x14, uint64_t x15,
         uint64_t x13, uint64_t x11) {
  uint64_t x17, uint64_t x18 = mulx_u64(x5, x11);
  // ...104 more similar lines...
  uint64_t x322 = cmovznz(x318, x305, x292);
  return (x319, x320, x321, x322))
}
```

Figure 1. Example input and output of code generation

at approximately the abstraction level of assembly. Furthermore, to achieve best performance, code is *written with particular hardware architectures in mind*. We show how to achieve similar high assurance levels while also achieving automatic compilation when changing the curve or target architecture.

Figure 1 gives a more concrete sense of what our framework provides, for generating custom modular-arithmetic code. The only input is a (usually large) prime number, written in a suggestive way with additions and subtractions, where most literals are powers of 2. The particular prime in the figure happens to be NISTP256, the most commonly used one for TLS.

Our framework uses the prime's addition-and-subtraction structuring to choose a data structure and algorithms (for different standard arithmetic operations). The figure shows part of the example of modular multiplication. The function takes in 8 inputs, as each big integer has been split into 4 word-sized digits, and we multiply 2 big integers. The body of the function is literally pretty-printed within Coq from an abstract syntax tree in a formal straightline-code language, really more like a compiler IR than C. The only additional features beyond standard C are for intrinsics and derived operations with multiple return values. A thin layer of scripting converts this literal Coq output into real GCC-compatible C code that uses nonstandard intrinsics for, e.g., multiplication generating two words of output. A Coq theorem is also generated, whose trusted base only includes the syntax and semantics of our straightline-code language plus standard arithmetic definitions.

The next section overviews our entire proof and code-generation pipeline, describing techniques that should apply beyond the concrete setting of ECC. The following three sections go into more detail on three key phases of the pipeline for ECC. Afterward, we discuss experimental evaluation, compare with related work, and conclude. Our framework source code and benchmarking examples and scripts are included as an anonymous supplement to the paper.

2 Outline of Compilation and Verification Pipeline

In this section, we run through all of the main steps in our compilation pipeline, on simpler examples than full-fledged cryptography primitives. We believe that our pipeline formalizes the procedures that crypto-implementation experts have been applying implicitly.

As we are generating code whose primary purpose is to promote security and privacy, a word is also in order about threat models and trusted code bases. In this project, when it comes to proved properties, we are concerned only with functional correctness: the low-level code we output implements a fixed mathematical function (the specification). It is also very important to avoid information leaks through side channels. Our code is designed to avoid timing side channels using the standard techniques of this domain, and the low-level language we use for generated straightline code only exposes functionality that is widely implemented in constant time in commodity hardware. Side channels requiring physical access (like those based on monitoring electromagnetic emissions) we leave out of scope. Also out of scope are proofs that the mathematical algorithms we implement provide standard security conditions from the theory of cryptography.

Our trusted code base includes the Coq proof checker and its usual dependencies. We also trust the (relatively small) functionality specifications sketched in the next subsection. At the back end of our pipeline, we have assembly-like abstract syntax trees that are proved to implement the original specifications. Currently we trust a C compiler used to translate those trees to assembly (after applying a trusted but small pretty-printer), though we expect eventually to integrate with a lower-level certified compiler.

2.1 The Specification

The fundamental objective of our work is to make it possible to write algorithms as straightforward programs (with some of the classic characteristics of "pseudocode") but have them compiled automatically to performance-competitive low-level code that is free of timing side channels. As a somewhat orthogonal bonus, we want machine-checked proofs that compilation is performed correctly. These goals taken together imply that it is reasonable to write starting specifications as functional programs in Coq. We also write example code in some unspecified functional language with lightweight syntax, as opposed to literal Coq syntax.

ECC is based on manipulation of points in two-dimensional geometric spaces, and we will work through an example sharing that property. We take some large prime modulus p as fixed throughout, and we write $\mathbb{N}_p$ for the modular-arithmetic field associated with p. Arithmetic operations are implicitly operating in that field.

$$\text{type } \mathbf{point} = \mathbb{N}_p \times \mathbb{N}_p$$

$$\mathbf{frob}\ ((x_1, y_1)\ (x_2, y_2) : \mathbf{point}) : \mathbf{point} = (x_1 + x_2, (y_1 \times y_2) \times x_1^{-1})$$

We define some arbitrary point operation **frob**, built out of addition, multiplication, and inversion. The level of simplicity in the code here is the standard we strive for.

2.2 Optimized Point Formats

One distinctive characteristic of this domain is that many algorithmic challenges can be tackled quite effectively in high-level functional code, even though we choose data structures and algorithms with an eye toward efficient execution on particular hardware platforms. Our first example of the pattern comes in selection of optimized point formats, i.e. data structures for our two-dimensional points. Field inversion, it turns out, is much more expensive than addition or multiplication. As a result, it is worthwhile to trade inversions for simpler operations, even at the expense of increasing the sizes of data structures. Our running **frob** example provides an opportunity for this kind of algorithmic rethinking.

Concretely, we make the counterintuitive choice of representing points with *three* coordinates each, instead of two. The intuition is that the new final coordinate gives a divisor to apply to the second coordinate.

$$\text{type } \mathbf{point} = \mathbb{N}_p \times \mathbb{N}_p \times \mathbb{N}_p$$

$$\mathbf{frob'}\ ((x_1, y_1, d_1)\ (x_2, y_2, d_2) : \mathbf{point}) : \mathbf{point} = (x_1 + x_2, y_1 \times y_2, d_1 \times d_2 \times x_1)$$

The payoff is that now no inversion operations are required for most computation steps.

We carry out classic data-abstraction proofs to show that optimized formats and their methods are faithful to simple formats. For this particular example, we prove the usual commuting diagrams with respect to this *abstraction function*:

$$\lfloor (x, y, d) \rfloor \quad \triangleq \quad \left(x, \frac{y}{d}\right)$$

The proof obligation for **frob** is:

$$\forall a, b.\ \lfloor \mathbf{frob'}\ a\ b \rfloor = \mathbf{frob}\ \lfloor a \rfloor\ \lfloor b \rfloor$$

Here the algebra is trivial. Full-scale elliptic curves require algebra complex enough that computer-algebra systems are routinely used to validate it. Our proofs duplicate that style of reasoning inside Coq, partly based on new tactics that we developed for this purpose, described in Section 3.

2.3 Base Systems for Multi-Digit Representation

Next on the agenda is implementing the numeric operators like + and × that still appear in our optimized point arithmetic. The numbers involved are typically too large to fit in single hardware registers, so we need to represent numbers explicitly as sequences of digits, each digit typically about the size of the largest available register. To start out with, let us consider the example of addition, with the simplifying

precondition that all digits are small enough to avoid the need to carry between them.

$$\begin{aligned}
&\text{type } \mathbf{num} = \mathbf{list}\ \mathbb{N}_p\\
&\mathbf{add} : \mathbf{num} \to \mathbf{num} \to \mathbf{num}\\
&\mathbf{add}\ (a :: as)\ (b :: bs) = \text{let } n = a + b \text{ in } n :: \mathbf{add}\ as\ bs\\
&\mathbf{add}\ as\ [] = as\\
&\mathbf{add}\ []\ bs = bs
\end{aligned}$$

Assume we are compiling for a 64-bit machine, where it is natural to make each digit a 64-bit integer. We define an abstraction function compiling each digit sequence (taken as little-endian) back into a single large number.

$$\lfloor \ell \rfloor \quad = \quad \Sigma_{i<|\ell|}\ \ell_i \times 2^{64i}$$

Next we can prove data-abstraction theorems similar to the ones from the prior subsection, one for each arithmetic operation. For instance, we prove the following for our addition operation.

$$\forall a, b.\ \lfloor \mathbf{add}\ a\ b \rfloor = \lfloor a \rfloor + \lfloor b \rfloor$$

One challenge in machine arithmetic is avoiding unintended overflow. However, our reasoning at this stage avoids explicit overflow reasoning by representing all digits as infinite-precision integers. Here we see another instance of the pattern of anticipating low-level optimizations in writing high-level code: we do expect to avoid overflow, and our choice of a digit representation is motivated precisely by that aim. It is just that the proofs of overflow-freedom will be injected in a later stage of our pipeline, as long as earlier stages like our current one are implemented correctly. There is good reason for not keeping overflow reasoning encapsulated in high-level stages: generally we care about the *context* of higher-level code calling our arithmetic primitives.

Section 4 presents the actual library of multi-digit arithmetic algorithms that we implemented and verified.

2.4 Partial Evaluation

It is impossible to achieve competitive performance with arithmetic code that manipulates dynamically allocated lists at runtime. The fastest code will implement, for instance, a single numeric addition with straightline code that keeps as much state as possible in registers. Expert implementers today write that straightline code manually, applying various rules of thumb. Our alternative is to use *partial evaluation* in Coq to generate all such specialized routines, beginning with a single library of high-level functional implementations.

Consider the case where we know statically that each number we add will have 3 digits. A particular addition in our top-level algorithm may have the form $\mathbf{add}\ [a_1, a_2, a_3]\ [b_1, b_2, b_3]$, where the a_is and b_is are unknown program inputs. While we cannot make compile-time simplifications based on the values of the digits, we *can* reduce away all the overhead of dynamic allocation of lists. We use Coq's term-reduction machinery, which allows us to choose λ-calculus-style reduction rules to apply until reaching a normal form. Here is what happens with our example, when we ask Coq to leave let expressions unreduced but apply most other rules.

$$\begin{aligned}
\mathbf{add}\ [a_1, a_2, a_3]\ [b_1, b_2, b_3] \quad \Downarrow \quad & \text{let } n_1 = a_1 + b_1 \text{ in } n_1 ::\\
& \text{let } n_2 = a_2 + b_2 \text{ in } n_2 ::\\
& \text{let } n_3 = a_3 + b_3 \text{ in } n_3 :: []
\end{aligned}$$

We have made progress: no run-time case analysis on lists remains. Unfortunately, let expressions are intermixed with list constructions, leading to code that looks rather different than assembly. Thus we come to another complication that we introduce to drive performant code generation: arithmetic operations are written in *continuation-passing style*. Concretely, we rewrite **add**.

$$\begin{aligned}
&\mathbf{add'} : \forall \alpha.\ \mathbf{num} \to \mathbf{num} \to (\mathbf{num} \to \alpha) \to \alpha\\
&\mathbf{add'}\ (a :: as)\ (b :: bs)\ k = \text{let } n = a + b \text{ in}\\
&\quad \mathbf{add'}\ as\ bs\ (\lambda \ell.\ k\ (n :: \ell))\\
&\mathbf{add'}\ as\ []\ k = k\ as\\
&\mathbf{add'}\ []\ bs\ k = k\ bs
\end{aligned}$$

Now Coq's normal reduction is able to turn our nice abstract functional program into assembly-looking code.

$$\begin{aligned}
\mathbf{add'}\ [a_1, a_2, a_3]\ [b_1, b_2, b_3]\ (\lambda \ell.\ \ell) \quad \Downarrow \quad & \text{let } n_1 = a_1 + b_1 \text{ in}\\
& \text{let } n_2 = a_2 + b_2 \text{ in}\\
& \text{let } n_3 = a_3 + b_3 \text{ in}\\
& [n_1, n_2, n_3]
\end{aligned}$$

When this procedure is applied to a particular continuation, we can reduce away the result list. We get attractive composition properties, where chaining together sequences of function calls leads to idiomatic and efficient assembly-style code, based just on Coq's normal term reduction, with good (and automatic) sharing of common subterms via let-bound variables. This level of function inlining is common for the inner loops of crypto primitives, and it will also simplify the static analysis described in the next subsection.

2.5 Bounds Inference

Up to this point, we have derived code that looks almost exactly like the assembly code we want to produce. The code is structured to avoid overflows when run with fixed-precision integers, though we are still using infinite-precision integers. The final major step is to infer a range of possible values for each variable, allowing us to assign each one a register or stack-allocated variable of the appropriate bit width.

This phase of our pipeline is systematic enough that we chose to implement it as a certified compiler. That is, we define a type of abstract syntax trees (ASTs) for the sorts of programs that earlier phases produce, we reify those programs into our AST type, and we run compiler passes written in Coq's Gallina functional programming language. Each pass is proved correct once and for all, as Section 5 explains in more detail.

The bounds-inference pass basically works by standard abstract interpretation with intervals. As inputs, we require

lower and upper bounds for the integer values of all free variables in a program. These bounds are then pushed through all operations in the program, to infer bounds for temporary variables. Each temporary is assigned the smallest bit width that can accommodate its full interval.

As an artificial example, assume the input bounds $a_1, a_2, a_3, b_1 \in [0, 2^{31}]$; $b_2, b_3 \in [0, 2^{30}]$. The analysis concludes $n_1 \in [0, 2^{32}]$; $n_2, n_3 \in [0, 2^{30} + 2^{31}]$. The first temporary is just barely too big to fit in a 32-bit register, while the second two will fit just fine. Therefore, assuming the available temporary sizes are 32-bit and 64-bit, we can transform the code with precise size annotations.

$$\begin{aligned}
&\text{let } n_1 : \mathbb{N}_{2^{64}} = a_1 + b_1 \text{ in} \\
&\text{let } n_2 : \mathbb{N}_{2^{32}} = a_2 + b_2 \text{ in} \\
&\text{let } n_3 : \mathbb{N}_{2^{32}} = a_3 + b_3 \text{ in} \\
&[n_1, n_2, n_3]
\end{aligned}$$

Note how we may infer different temporary widths based on different bounds for the free variables. As a result, the same primitive inlined within different larger procedures may get different bounds inferred. World-champion code for real algorithms takes advantage of this opportunity.

2.6 Generating Assembly-Like Code

We finish with ASTs in a simple language of straightline code, with arithmetic and bitwise operators. Our future-work plans include creating enough Coq certifying-compilation support to handle surrounding code with loops and conditionals, but we have also run some performance experiments that are already feasible. We take the ASTs of our generated arithmetic primitives and pretty-print them as C code, benchmark them separately, or overwrite the corresponding code in popular C implementations. Section 6 reports on our performance experiments, but a good summary is that we are 5X faster than generic multi-precision arithmetic libraries, faster than OpenSSL cross-platform C code, and within 2X of world-champion handwritten assembly code.

We now use the bulk of the paper to go back through the phases of our compilation in more detail, before saying more about the specific primitives we have generated and the experiments we ran on our implementations.

3 Curve Data Structures and Algorithms

The main reusable methodology we want to highlight in this paper is for correct-by-construction generation of efficient low-level code for modular big-number arithmetic. However, we also built complete implementations of ECC-based key exchange, signing, and (signature) verification, parameterized on arithmetic implementations. Since our specification and proof choices there are interestingly different than in past work, we say a bit about them here. Connecting our modular-arithmetic proofs to end-to-end arguments about complete primitives gives us confidence that we chose the right theorems to prove about modular arithmetic.

Recall Section 2.1, giving a toy example of a geometric point type and one of its operations. Elliptic curves are all about more involved point types and operations. Recall also Section 2.2, which performed a change of data representation for points. A menagerie of standard representation changes exists for elliptic curves: we defined and verified affine, XYZT, and Niels variants of Edwards coordinates; affine, Jacobian, and Projective Weierstrass coordinates; and affine and XZ Montgomery coordinates.

Past related work we are aware of (e.g. Zinzindohoue et al. [21]) has only taken the already-optimized point formats as the starting specification. By starting with the more elementary formats, we simplify specifications and decrease trusted base. These optimizations are nontrivial. Even experts need to apply computer-algebra systems to check all the details. Often optimized algorithms are only sound for particular subsets of curve points, and higher-level algorithm proofs must show that corresponding preconditions are always met. We formalized preconditions for all the operations of all the optimized point formats and proved them sufficient.

To prove the operations correct, we need functionality similar to that provided by computer-algebra systems like Sage. We build upon the `nsatz` [16] tactic from Coq's standard library, which solves implications between polynomial equalities. Our tactic `fsatz` broadens the scope to high-school-algebra examples like this one: given $\frac{9}{x^2+x-2} = \frac{3}{x+2} + 7\frac{1}{x-1}$ and appropriate assumptions about the coefficients and denominators being nonzero, we may deduce $x = -\frac{1}{5}$. Efficient support is particularly important for using and proving inequalities, as required for each denominator in the goal.

Through a set of heuristics for reducing arithmetic operators and relations to more elementary ones, we produce `nsatz`-compatible goals and manage to prove all the key point-format properties quickly and predictably. For example, `fsatz` solves all 131 field equations (a total of 72 kB of text) required for a direct proof that every elliptic curve in Weierstrass form is a commutative group.

4 Generic Modular Arithmetic

After we commit to particular optimized point formats, attention turns to the numeric operations of the prime field, used to compute individual coordinates of points. Recall Section 2.3's example of custom code implementing a numeric base system. We now describe our full-scale library.

For those who prefer to read code, we suggest `src/Demo.v` in the code supplement to this submission, which contains a succinct standalone development of the unsaturated-arithmetic library up to and including modular reduction.

4.1 Multi-Limbed Arithmetic

Before describing our library, we review the motivation and algorithmic big ideas of this style of arithmetic. The first piece of motivation is shared with conventional big-integer

libraries: a single integer is too large to fit in a hardware register, so we must represent one big integer with several smaller *digits* (often called *limbs* in the crypto context). The interesting difference is in how subtle it is to design a strategy for dividing a number into digits; as we will show, this choice depends heavily on the particular prime modulus being used.

The most popular choices of primes in elliptic-curve cryptography are of the form $m = 2^k - c_l 2^{t_l} - \ldots - c_0 2^{t_0}$, encompassing what have been called "generalized Mersenne primes," "Solinas primes," "Crandall primes," "pseudo-Mersenne primes," and "Mersenne primes." Although any number could be expressed this way, and the algorithms we describe would still apply, choices of m with relatively few terms ($l \ll k$) and small c_i more readily facilitate fast arithmetic.

Imagine that we have two numbers that are about the same size as the modulus (k bits), and we multiply them. We would need $2k$ bits to represent the result. However, we only care about what the result is mod m. So we apply a (partial) *modular reduction*, an operation that reduces the upper bound on its input while preserving modular equivalence.

With this form of prime, there is a well-known trick for simple and fast modular reduction. Set $s = 2^k$ and $c = c_l 2^{t_l} + \ldots + c_0 2^{t_0}$, so $m = s - c$. To reduce x mod m, first find a and b such that $x = as + b$. (We call this operation `split`, and careful choices of big-number representation will make it very efficient.) Then a simple derivation yields a division-free procedure for partial modular reduction:

$$\begin{aligned} x \bmod m &= (as + b) \bmod (s - c) \\ &= (a(s - c) + ac + b) \bmod (s - c) \\ &= (ac + b) \bmod m \end{aligned}$$

The choice of a and b does not further affect the correctness of this formula, but it does influence how much the input is reduced: picking $b = x$ and $a = 0$ would make this formula a no-op. One might pick $b = x \bmod s$, although the formula does not require it. Even if $b = x \bmod s$, the final output $ac + b$ is not guaranteed to be the minimal residue.

Making the `split` operation fast will motivate how we represent numbers. Consider Curve25519 ($m = 2^{255} - 19$, $k = 255$), where an intermediate multiplication result requires 510 bits. One natural way to represent it uses 8 64-bit registers, like so, where t_i is the ith digit/register:

$$(t_0 + 2^{64}t_1 + 2^{2\times64}t_2 + 2^{3\times64}t_3) + 2^{256}(t_4 + 2^{64}t_5 + 2^{2\times64}t_6 + 2^{3\times64}t_7)$$

We split the digit sequence in half suggestively, such that the values of the two sides can be combined using a multiplication by 2^{256}. If 2^{256} were 2^{255}, we could have our `split` operation entirely "for free" – this formula is already in the form $b + 2^{256}a$. Unfortunately, 256 is not 255, and the property does not apply! This off-by-one error motivates a rather different strategy for dividing a number into digits.

Instead, we could divide 510 bits into 10 groups of 51 bits each. That is, we will use 64-bit registers but not even take advantage of the full value space for each one. Now we get a more satisfying formula to convert back into one big number.

$$\begin{aligned} &(t_0 + 2^{51}t_1 + 2^{2\times51}t_2 + 2^{3\times51}t_3 + 2^{4\times51}t_4) \\ &+ 2^{255}(t_5 + 2^{51}t_6 + 2^{2\times51}t_7 + 2^{3\times51}t_8 + 2^{4\times51}t_9) \end{aligned}$$

The 2^{255} lets us apply the modular-reduction optimization. This representation is standard for 64-bit processors, found in essentially every major crypto library and Web browser.

That is not the end of the story for this curve, though. On 32-bit machines, we do better with a representation that fits in 32-bit registers. The best-performing solution divides the 510 bits into 20 groups of 25.5 bits each, or actually we use a ceiling operation to round each such bit width. The 32-bit registers for digits alternate between getting 26 and 25 bits each, which happens to line us up for a 2^{255} in just the right place. We have a *mixed-radix* base, as opposed to a *uniform-radix* base in which every digit has the same number of bits. This odd-seeming data structure appears in the 32-bit versions of the major crypto libraries and browsers.

Already, then, for this important prime modulus, we see three different well-justified representations. Different hardware platforms could imply still more representations. It would behoove us to find code-reuse (and proof-reuse) opportunities that quantify over the essence of the different representations.

Following that strategy, we also need to implement generic algorithms that adapt to different digit decompositions. We will illustrate with just one key algorithm specialized to just one modulus and digit strategy. To simplify matters a bit, we use modulus $2^{127} - 1$. Say we want to multiply 2 numbers s and t in its field, with those inputs broken up as $s = s_0 + 2^{43}s_1 + 2^{85}s_2$ and $t = t_0 + 2^{43}t_1 + 2^{85}t_2$. Distributing multiplication repeatedly over addition gives us the answer form shown in Figure 2.

We format the first intermediate term suggestively: down each column, the powers of two are very close together, differing by at most one. Therefore, it is easy to add down the columns to form our final answer, split conveniently into digits with integral bit widths.

At this point we have a double-wide answer for multiplication, and we need to do modular reduction to shrink it down to single-wide. For our example, note that the last two digits can be rearranged like so:

$$\begin{aligned} & 2^{127}(2s_1t_2 + 2s_2t_1) + 2^{170}s_2t_2 \pmod{2^{127} - 1} \\ = \; & 2^{127}((2s_1t_2 + 2s_2t_1) + 2^{43}s_2t_2) \pmod{2^{127} - 1} \\ = \; & 1((2s_1t_2 + 2s_2t_1) + 2^{43}s_2t_2) \pmod{2^{127} - 1} \end{aligned}$$

As a result, we can merge the second-last digit into the first and merge the last digit into the second, leading to this final formula for a single-width answer.

$$(s_0t_0 + 2s_1t_2 + 2s_2t_1) + 2^{43}(s_0t_1 + s_1t_0 + s_2t_2) + 2^{85}(s_0t_2 + 2s_1t_1 + s_2t_0)$$

We still manage to restrict ourselves to a modest number of elementary arithmetic operations. Also, there are not many

$$
\begin{array}{lllll}
s \times t & = 1 \times s_0 t_0 & + 2^{43} \times s_0 t_1 & + 2^{85} \times s_0 t_2 & \\
 & & + 2^{43} \times s_1 t_0 & + 2^{86} \times s_1 t_1 & + 2^{128} \times s_1 t_2 \\
 & & & + 2^{85} \times s_2 t_0 & + 2^{128} \times s_2 t_1 \quad + 2^{170} \times s_2 t_2 \\
 & = s_0 t_0 & + 2^{43}(s_0 t_1 + s_1 t_0) & + 2^{85}(s_0 t_2 + 2 s_1 t_1 + s_2 t_0) & + 2^{127}(2 s_1 t_2 + 2 s_2 t_1) \quad + 2^{170} s_2 t_2
\end{array}
$$

Figure 2. Distributing terms for multiplication mod $2^{127} - 1$

data dependencies within the expression, so there are good opportunities for instruction-level parallelism on modern processors.

4.2 Further Challenges

We do not have space to explain the full range of additional wrinkles that show up in deriving all of the common code patterns for modular arithmetic in ECC. However, here are some highlights.

- Different combinations of moduli and hardware architectures are suited to *saturated* vs. *unsaturated* arithmetic, where the former uses the full bitwidth of hardware registers, and the latter leaves bits unused.
- All of our examples above used primes of the form $2^k - c$ where c was very small. In those cases, computing $ac + b$ on multi-digit integers is reasonably straightforward: multiply each digit of a by c and add each digit of the result ac to the corresponding digit of b. Because we are not using the full bit widths of our registers, and because c is quite small, overflow is not even an issue. However, the same formula applies for larger c, such as in NIST p-192 ($m = 2^{192} - 2^{64} - 1$). Now we ought to perform multi-digit multiplication of a and c – working very similarly to polynomial multiplication.
- In unsaturated base systems, by design we are not carrying immediately after every addition. Therefore, choosing when and which digits to carry is part of the design and is critical for keeping the digit values bounded. Generic operations are easily parameterized on carry strategies, although our library uses a conservative heuristic by default.

4.3 Associational Representation

As is evident by now, the most efficient code makes use of sophisticated and specific big-number representations, but all of these tend to operate on the same set of underlying principles. We want to reason about the basic arithmetic procedures (multiplication, carrying, modular reduction) in a way that allows us access to those underlying principles while abstracting away implementation-specific details like the exact number of limbs or whether the base system is mixed- or uniform-radix. Designing our system such that this level of reasoning was possible was one of the key factors in making our verification successful.

Our initial attempt at formalizing mixed-radix base systems involved keeping track of two lists, one with the base weights (i.e., power of 2 associated with each digit) and one with the corresponding runtime values. This version was very messy; we had to keep track of preconditions stating that the lists had the same length, and in basic arithmetic operations we were constantly dealing with the details of the base. For instance, in multiplication, every time we obtained a partial product, we had to check if the weight of the partial product matched one of our fixed digit weights (not guaranteed with mixed-radix bases) and, if not, shift the partial product before inserting it into the right place in the list. That representation was very close to how things were written in the C code; however, it was not the best way to represent the algorithms conceptually, and it introduced unnecessary complexity.

In our second attempt, we came up with what we call *associational* representation–a list of pairs, where one number represents the weight, known at compile time, and the other represents a runtime value. For example, the decimal number 95 might be encoded as `[(10, 9); (1, 5)]` or `[(16, 5); (1, 15)]`, representing $10 \cdot 9 + 1 \cdot 5 = 16 \cdot 5 + 1 \cdot 15 = 95$. In an associational setting, proving multiplication, addition, and reduction became extremely straightforward. Addition is simply concatenating two lists. Schoolbook multiplication is also trivial: $(a_1 \cdot x_1 + \ldots)(b_1 \cdot y_1 + \ldots) = (a_1 b_1 \cdot x_1 y_1 + \ldots)$, where $a_1 b_1$ is a constant term that can be computed during partial evaluation. The details of the three fit in 6 lines of executable code, 4 lines of lemma statements, and 10 lines of proof (as written in `src/Demo.v`). The `split` step of modular reduction simply partitions the list into terms with weights higher than s and terms with weights lower than s, and then the rest of modular reduction just calls addition and multiplication.

However, we ultimately want to add the partial products and end up with one term per digit, in what we call a *positional* representation. We can convert from associational to positional using a weight function (importantly, we do not try to infer the weights from the associational representation). Weights that are present in the input but not in the desired positional representation are eliminated by multiplying the corresponding digit by a constant: converting $[(20, 3); (1, 7)]$ to a 2-digit base-10 representation yields 67 because $(20/10) \cdot 3 = 6$.

We then exposed the same positional interface as in our first attempt by simply converting to associational, performing whatever operations we needed, and converting back to positional. The change produced no clutter in our final output, since as soon as the base system and weight function are instantiated, the representation differences and conversions between them can be evaluated away.

Furthermore, representing things this way made our implementations generalize naturally. While in our first attempt we had only implemented modular reduction for very small c, the natural way to write the algorithm in associational representation is to represent c as a list of pairs and multiply it by a using the full Cartesian-product strategy. This strategy naturally generalizes to c with multiple terms, with no extra effort in code or proofs. Surprisingly, even to us when we first implemented it, this 5-line implementation is flexible enough to allow expressing any specialized modular-reduction-algorithm formula we know of – and the 15-line correctness proof applies to all of them. The design freedom comes from being able to choose different associational representations for c. For example, the prime modulus of the `secp256k1` elliptic curve used in Bitcoin, $2^{256} - 2^{32} - 977$ with $s = 2^{256}$, can be implemented reasonably using either $c = [(2^{32}, 1); (1, 977)]$ or $c = [(1, 2^{32} + 977)]$. The first option generates twice as many digit multiplications as the second but is still preferable on some architectures because all these partial products fit in 64 bits. On architectures such as AMD64 that can multiply two 64-bit numbers to get a 128-bit product, the second option has an advantage.

4.3.1 Saturated Arithmetic and Montgomery Modular Multiplication

However, in some cases, the base being used does warrant changes to the underlying arithmetic routines, most notably for saturated versus unsaturated representations. In unsaturated code, for instance, it is not necessary to worry about producing hardware instructions that set carry flags, but in saturated representations it is essential. Also, in unsaturated representations, we store the partial products in multiplication routines in double-wide registers, which makes sense, given that it does not help us to split the product along 64-bit boundaries (we would prefer the low 51 bits, for instance) and would require bit-shifting anyway. It is our experience that algorithms based on unsaturated representations are significantly easier to implement and reason about. However, while unsaturated arithmetic is very fast for X25519 and X448, every implementation of NISTP256 that achieves even remotely competitive performance uses as few machine registers as possible, relies on hardware instructions that are not readily exposed in most programming languages (like two-output multiplication and add-with-carry), and uses algorithms that require intermediate values to be within specific ranges. So when we decided to target that prime, it was necessary to implement an extension to our arithmetic routines.

Again, associational representation is helpful here. Our multiplication routine remained virtually the same, the only change being that instead of producing (ab, xy) as the partial product for terms (a, x) and (b, y), we now produce `let xy := mul x y in [(ab, fst xy); (ab * bound, snd xy)]`, where bound is the size of the registers. This new form of partial product could be appended to the rest of the list and thenceforth handled using literally the same code as we had used for unsaturated representations; for instance, there was no need to change the code for modular reduction. Even addition used the same code, since associational representation does not require us to add terms together and worry about carries just yet.

Instead, we worried about carries only when converting from associational to positional. We created an intermediate representation (again, leveraging our ability to switch between whatever representations are convenient) that accumulated terms at each position without adding them. Then we could do an addition loop for each weight, repeatedly adding up the terms of the smallest remaining weight and accumulating their carries into one (multi-bit) term. The carry term would then be added to the next weight.

The takeaway here is that even completely changing the underlying hardware instructions we used for basic arithmetic did not require redoing all the work from unsaturated representations.

Our most substantial use of saturated arithmetic was for *Montgomery modular reduction*. In some circumstances, computing $ab \bmod m$ is rather expensive. Instead, we replace all intermediate values x with xR, multiplying by some fixed weight R. Such values are said to be in Montgomery form. Now imagine we have a fast way, given a and b, to calculate $abR^{-1} \bmod m$. When a and b are really $a'R$ and $b'R$, the result of the operation is $(a'R)(b'R)R^{-1} \bmod m = (a'b')R \bmod m$, which conveniently returns to Montgomery form.

5 Certified Bounds Inference

Recall from Section 2.4 how we use partial evaluation to specialize the functions from the last section to particular parameters. The results are elementary enough code that it becomes more practical to apply relatively well-understood ideas from *certified compilers*. That is, as sketched in Section 2.5, we can define an explicit type of program abstract syntax trees (ASTs), write compiler passes over it as Coq functional programs, and prove those passes correct once and for all.

5.1 Abstract Syntax Trees

The results of partial evaluation fit, with minor massaging, into this intermediate language that we defined.

$$\begin{array}{rlll} \text{Base types} & b & & \\ \text{Types} & \tau & ::= & b \mid \mathsf{unit} \mid \tau \times \tau \\ \text{Variables} & x & & \\ \text{Operators} & o & & \\ \text{Expressions} & e & ::= & x \mid o(e) \mid () \mid (e, e) \\ & & & \mid \mathsf{let}\ (x_1, \ldots, x_n) = e\ \mathsf{in}\ e \end{array}$$

Types are trees of pair-type operators $\times$ where the leaves are one-element unit types and base types b, the latter of which come from a domain that is a parameter to our compiler. It will be instantiated differently for different target hardware architectures, which may have different primitive integer types. When we reach the certified compiler's part of the pipeline, we have converted earlier uses of lists into tuples, so we can optimize away any overhead of such value packaging.

Also a language parameter is the set of available primitive operators o, each of which takes a single argument, which is often a tuple of base-type values. Our let construct bakes in destructuring of tuples, in fact using typing to ensure that all tuple structure is deconstructed fully, with variables bound only to the base values at a tuple's leaves. Our deep embedding of this language in Coq uses dependent types to enforce that constraint, along with usual properties like lack of dangling variables and type agreement between operators and their arguments.

Several of the key compiler phases are polymorphic in the choices of base types and operators, but bounds inference is specialized to a set of operators. We assume that each of the following is available for each type of machine integers (e.g., 32-bit vs. 64-bit).

Integer literals: n
Unary arithmetic operators: $-e$
Binary arithmetic operators: $e_1 + e_2, e_1 - e_2, e_1 \times e_2$
Bitwise operators: $e_1 \ll e_2, e_1 \gg e_2, e_1 \,\&\, e_2, e_1 \mid e_2$
Conditionals: $\mathsf{if}\ e_1 \neq 0\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3$
Carrying: $\mathsf{addWithCarry}(e_1, e_2, c), \mathsf{carryOfAdd}(e_1, e_2, c)$
Borrowing: $\mathsf{subWithBorrow}(c, e_1, e_2), \mathsf{borrowOfSub}(c, e_1, e_2)$
Two-output multiplication: $\mathsf{mul2}(e_1, e_2)$

We explain the last three categories, since the earlier ones are familiar from C programming. To chain together multi-word additions, as discussed in the prior section, we need to save overflow bits (i.e., carry flags) from earlier additions, to use as inputs into later additions. The addWithCarry operation implements this three-input form, while carryOfAdd extracts the new carry flag resulting from such an addition. Analogous operators support subtraction with *borrowing*, again in the grade-school-arithmetic sense. Finally, we have mul2 to multiply two numbers to produce a two-number result, since multiplication at the largest available word size may produce outputs too large to fit in that word size.

All operators correspond directly to common assembly instructions. Thus the final outputs of compilation look very much like assembly programs, just with unlimited supplies of temporary variables, rather than registers.

$$\begin{array}{rlll} \text{Operands} & O & ::= & x \mid n \\ \text{Expressions} & e & ::= & (O, \ldots, O) \\ & & & \mid \mathsf{let}\ (x_1, \ldots, x_n) = o(O, \ldots, O)\ \mathsf{in}\ e \end{array}$$

We no longer work with first-class tuples. Instead, programs are sequences of primitive operations, applied to constants and variables, binding their perhaps multiple results to new variables. A function body, represented in this type, ends in the function's perhaps multiple return values.

Such functions are easily pretty-printed as C code, which is how we compile them for our experiments. Note also that the language enforces the *constant time* security property by construction: the running time of an expression leaks no information about the values of the free variables. (One additional restriction is important, forcing conditional expressions to be those supported by native processor instructions like conditional move.)

5.2 Phases of Certified Compilation

To begin the certified-compilation phase of our pipeline, we need to *reify* native Coq programs as terms of this AST type. To illustrate the transformations we perform on ASTs, we walk through what the compiler does to an example program:

$$\begin{array}{l} \mathsf{let}\ (x_1, x_2, x_3) = x\ \mathsf{in} \\ \mathsf{let}\ (y_1, y_2) = ((\mathsf{let}\ z = x_2 \times 1 \times x_3\ \mathsf{in}\ z + 0), x_2)\ \mathsf{in} \\ y_1 \times y_2 \times x_1 \end{array}$$

The first phase is *linearize*, which cancels out all intermediate uses of tuples and immediate let-bound variables and moves all lets to the top level.

$$\begin{array}{l} \mathsf{let}\ (x_1, x_2, x_3) = x\ \mathsf{in} \\ \mathsf{let}\ z = x_2 \times 1 \times x_3\ \mathsf{in} \\ \mathsf{let}\ y_1 = z + 0\ \mathsf{in} \\ y_1 \times x_2 \times x_1 \end{array}$$

Next is *constant folding*, which applies simple arithmetic identities and inlines constants and variable aliases.

$$\begin{array}{l} \mathsf{let}\ (x_1, x_2, x_3) = x\ \mathsf{in} \\ \mathsf{let}\ z = x_2 \times x_3\ \mathsf{in} \\ z \times x_2 \times x_1 \end{array}$$

At this point we run the core phase, *bounds inference*, the one least like the phases of standard C compilers. The phase is parameterized over a list of available fixed-precision base types with their ranges; for our example, assume the hardware supports bit sizes 8, 16, 32, and 64. Intervals for program inputs, like x in our running example, are given as additional inputs to the algorithm. Let us take them to be as follows:

$x_1 \in [0, 2^8], x_2 \in [0, 2^{13}], x_3 \in [0, 2^{18}]$. The output of the algorithm has annotated each variable definition and arithmetic operator with a finite type.

$$\text{let } (x_1 : \mathbb{N}_{2^{16}}, x_2 : \mathbb{N}_{2^{16}}, x_3 : \mathbb{N}_{2^{32}}) = x \text{ in}$$
$$\text{let } z : \mathbb{N}_{2^{32}} = x_2 \times_{\mathbb{N}_{2^{32}}} x_3 \text{ in}$$
$$z \times_{\mathbb{N}_{2^{64}}} x_2 \times_{\mathbb{N}_{2^{64}}} x_1$$

Our biggest proof challenge here was in the interval rules for bitwise operators applied to negative numbers, a subject mostly missing from Coq's standard library.

5.3 Important Design Choices

Most phases of the compiler use a term encoding called parametric higher-order abstract syntax (PHOAS) [9]. Briefly, that encoding uses variables of the metalanguage (Coq's Gallina) to encode variables of the object language, to avoid most kinds of bookkeeping about variable environments; and for the most part we found that it lived up to that promise. However, we needed to convert to a first-order representation (de Bruijn indices) and back for the bounds-inference phase, essentially because it calls for a forward analysis followed by a backward transformation: calculate intervals for variables, then rewrite the program bottom-up with precise base types for all variables. We could not find a way with PHOAS to write a recursive function that returns both bounds information and a new term, taking better than quadratic time, while it was trivial to do with first-order terms. We also found that the established style of term well-formedness judgment for PHOAS was not well-designed for large, automatically generated terms like ours: proving well-formedness would frequently take unreasonably long, as the proof terms are quadratic in the size of the syntax tree. The fix was to switch well-formedness from an inductive definition into an executable recursive function that returns simple constraints in propositional logic.

6 Experimental Results

Our framework has a straightforward story for formal guarantees of functional correctness, thanks to the use of Coq. However, some other important questions should be answered empirically: For a variety of prime moduli, how does the performance of our generated code compare with general-purpose arithmetic libraries? For the most popular primes, how far off performance-wise is our generated code from the best known implementations, and what accounts for the gap? Is the implementation and proof effort reasonable, to build a framework like ours? (This last question's answer we push to Appendix C.)

6.1 Automatic Code Generation for Many Primes

Recall that the whole framework package, presented in the previous few sections, is meant to be used to generate new ECC primitive routines automatically for new prime moduli (new curves). The output of the final phase from the last section is pretty-printed as C code and compiled with off-the-shelf C compilers. Our experiments instantiate the framework for different popular moduli, in each case benchmarking key arithmetic routines. Here we go for breadth, doing automatic compilation for all large primes scraped from the archives of `curves@moderncrypto.org`, an active ECC discussion forum[1]. We implement the key cryptographic operation of a 256-bit *Montgomery ladder* for each one, comparing our automatically generated code against simple parameterized C code, whose interpretation/compilation does not take advantage of number-theoretic optimizations keyed off of the prime modulus. Instead, code is just recompiled with a preprocessor macro set for the prime in question.

First, a simple Python script (under 300 lines of code) parses the prime and generates input files with additional parameters like which operations to synthesize, how to distribute field elements across smaller machine words, and modular-reduction strategy. Some heuristic complexity is embodied in the script, but bugs in it cannot compromise soundness, just lead to failed compilation attempts, which manifest as errors or timeouts in automatic Coq derivation.

Figure 3 shows the results of our experiments, demonstrating how running time scales with the number of bits needed to represent a number modulo the chosen prime. For each configuration, we compare our generated code with two variants built using the GNU Multiple Precision Arithmetic Library[2]. "GMP mpn" tests use a more performant API that leaks numeric values through timing, while "GMP mpn_sec" sacrifices performance for reduced leakage. Both versions require C-language expertise to code, though only one program each must be written, parameterized on a prime.

Our batch 64-bit trials run on an x86 Intel Haswell processor, while 32-bit trials run on an ARMv7-A Qualcomm Krait (LG Nexus 4). Benchmark time is measured for 1000 sequential computations of this operation. For each configuration, we show whichever of our two synthesized strategies (Solinas vs. Montgomery) gives better performance. We see a significant performance advantage for our code, even compared to the GMP version that "cheats" by leaking secrets through timing. Speedups range between 1.25X and 10X.

In our current experiments, compilation in Coq times out for a handful of larger primes; we continue to work on compile-time performance improvements. Appendix B includes the full details, with tables recording all experimental data points, including with an additional comparison implementation in C++.

6.2 X25519 Scalar Multiplication

The next benchmark tests our unsaturated-arithmetic synthesis against best-known open-source code. A single benchmark consists of an entire X25519 scalar multiplication to

[1] Archives: https://moderncrypto.org/mail-archive/curves/
[2] https://gmplib.org/

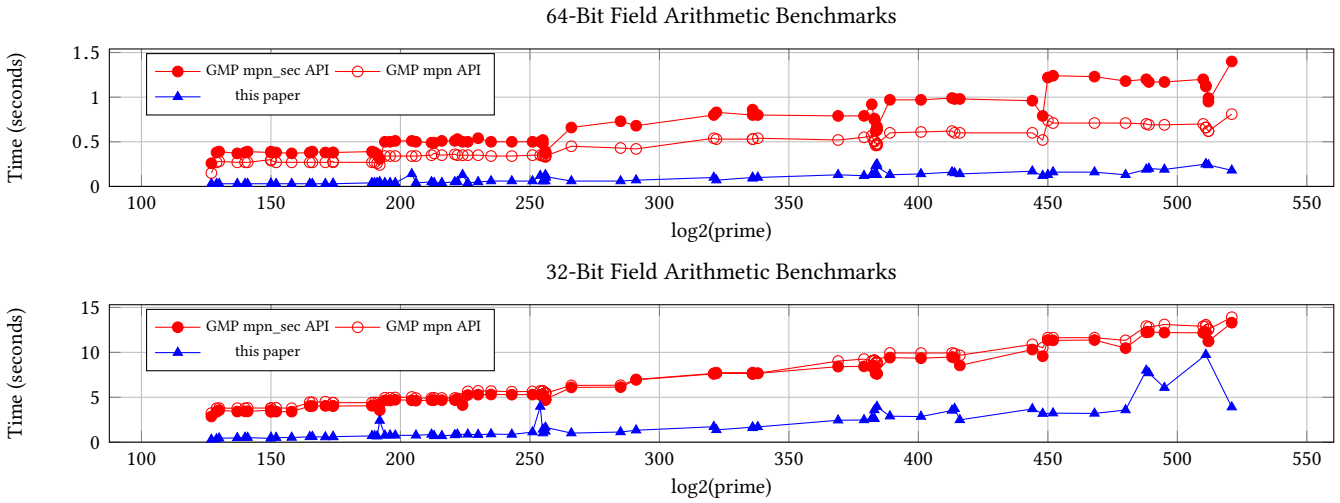

Figure 3. Performance comparison of our generated C code vs. handwritten using `libgmp`

make the benchmark end-to-end. However, all implementations that we compare against use the same scalar-multiplication algorithm; the differences are due to elliptic curve formulas and field arithmetic implementation. To gain insight into which further optimizations might be profitable to add to our framework, we measure both the literal output of our compiler and a hand-modified version, inspired by inspecting widely used assembly code, that we prove equivalent.

Implementation	CPU cycles	µs at 2.6GHz
`amd64-64 asm`	145008	56
`donna-c64` C	160352	62
this work, tweaked C	168364	65
this work, generated C	182580	70
OpenSSL C	348072	134

In order, we compare against `amd64-64 asm`, the fastest assembly implementation from SUPERCOP; `donna-c64`, the best-known high-performance C implementation; and OpenSSL's cross-platform C implementation. (Most of the names are official ones from the SUPERCOP benchmark suite [4].)

Both implementations from this work have correctness proofs of the same strength. The generated implementation is derived automatically from high-level templates with the minimal curve-specific parameters explained in Section 6.1. The tweaked implementation differs from the generated one as follows. We first executed the high-level stages of our pipeline, up to and including partial evaluation. Then we replaced the code for field-element squaring and multiplication with 30 lines of hand-written versions modeled after the `donna` implementation, proving it equivalent to the generated code using a single tactic invocation. After that, the pipeline continued through the lower-level phases as usual.

We believe the manual optimizations improved the performance because expressions of the form $19 \times (a \times b)$ with 32-bit a and b were replaced with $(19 \times a) \times b$ in cases where $19 \times a$ fits in 32 bits, reducing the number of 64-bit multiplications. Achieving this result without duplicating computation required careful reassociation and factoring of computations. With these manual tweaks, the compiled binaries for inner loops of our implementation and `donna` contain the same number of bitwise or arithmetic instructions (approximately 1000), but `donna` requires around 200 fewer moves. We leave the remaining 5% performance difference for low-level compiler work to fix up.

These results were good enough to convince the maintainers of Google Chrome to adopt our compiler for producing their Curve25519 code, within their BoringSSL library. For their preexisting Curve25519 benchmarks, we never lose on latency by worse than 5%, usually significantly less; Appendix D gives details. Our code was first adopted in Chrome version 64.

6.3 NISTP256 Mixed Addition

Our final performance experiment benchmarks our synthesized saturated arithmetic code. A single benchmark consists of one mixed Jacobian-Affine addition of distinct points on the NISTP256 curve.

Implementation	CPU cycles	µs at 2.6GHz
OpenSSL AMD64+ADX asm	544	.21
OpenSSL AMD64 asm	644	.25
this work, `icc`	1112	.43
this work, `gcc`	1808	.70
OpenSSL C	1968	.76

Our C code and the two assembly-language implementations from OpenSSL use the same overall implementation strategy: saturated arithmetic on 4 64-bit limbs using Montgomery multiplication. The two assembly-language implementations differ in what CPU features they require: the

slower is targeted at general AMD64 processors; the faster uses the ADX instruction-set extension that is available starting with Intel Broadwell (which we used) and AMD Zen microarchitectures, allowing for limited instruction-level parallelism in code that uses carry-flag registers. As arithmetic operations in our C source line up very closely with those in the OpenSSL assembly code and the difference between `icc` and `gcc` is bigger than the difference between our code and the fastest assembly code, we again attribute most of the performance difference to low-level optimizations.

7 Related Work

Several projects with papers published in mid-to-late 2017 have done formal verification of performance-competitive, low-level elliptic-curve code.

Vale [7] supports compile-time metaprogramming of assembly code, with a cleaner syntax to accomplish the same tasks done via Perl scripts in OpenSSL. There is a superficial similarity to the flexible code generation used in our own work. However, Vale and OpenSSL use comparatively shallow metaprogramming, essentially just doing macro substitution, simple compile-time offset arithmetic, and loop unrolling. Vale has not been used to write code parameterized on a prime modulus (and OpenSSL includes no such code). A verified static analysis checks that assembly code does not leak secrets, including through timing channels.

HACL* [22] is a cryptographic library implemented and verified in the F* programming language, providing all the functionality needed to run TLS 1.3 with the latest primitives. Primitives are implemented in the Low* imperative subset of F* [17], which supports automatic semantics-preserving translation to C. As a result, while taking advantage of F*'s high-level features for specification, HACL* beats or comes close to performance of leading C libraries. Additionally, abstract types for secret data rule out side-channel leaks.

Jasmin [1] is a low-level language that wraps assembly-style straightline code with C-style control flow. It has a Coq-verified compiler to 64-bit x86 assembly (with other targets planned), along with support for verification of memory safety and absence of information leaks, via reductions to Dafny. A Dafny reduction for functional-correctness proof exists but has not yet been used in a significant case study.

Several commonalities arise in comparing with our work.

Genericity in prime modulus: Our pipeline supports push-button generation of efficient code for new prime moduli. These other projects require nontrivial per-modulus work in implementation, e.g. to implement modular reduction; and specification/proof, e.g. to annotate every function with specialized integer-range preconditions and postconditions. Perhaps as a consequence of the work required to add a new modulus, the three projects taken together only implement intricate big-integer arithmetic for Curve25519 and Poly1305. The NISTP256 curve (among those that we generate) is both the most widely used (as of now) and significantly more involved to implement efficiently.

Genericity in target hardware architecture: All three projects include minimal code reuse across hardware architectures, without sacrificing performance-competitiveness. Every piece of code going into a primitive implementation has built-in an assumption about some target architecture. In contrast, we demonstrate full code/proof reuse between 32-bit ARM and 64-bit x86 targets.

Going beyond straightline code: For now, we only derive straightline code, where the constant-time security property holds by construction. Effectively, we focus on the performance-critical inner loops of cryptographic primitives. However, it would be valuable to expand our scope to generate and reason about the additional code around the inner loops, at which point it could make sense to connect to any of these three projects.

Lowering guarantees to assembly: Our current results bottom out in C-like programs, and it could be advantageous for us to connect to Jasmin or Vale to derive theorems about genuine assembly code. In fact, every one of our compiler phases is necessary to get code low-level enough to be accepted as input by any of the three other projects.

A few other projects have verified ECC code that must be handwritten in advance. Chen et al. [8] verified an assembly implementation of Curve25519, using a mix of automatic SAT solving and manual Coq proof for remaining goals. Bernstein and Schwabe [5] explored an alternative workflow using the Sage computer-algebra system. In a predecessor system to HACL*, Zinzindohoue et al. [21] verified more curves, including P256, but in high-level F* code, incurring performance overhead above 100X.

Performance-oriented synthesis of domain-specific code (without proofs of correctness) has previously been done using explicit templates (e.g. Template Haskell [20]) and more sophisticated multistage programming (e.g. Lightweight Modular Staging (LMS) [19]). More specialized frameworks along these lines include FFTW [10] and Spiral [18]. 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 [13]. 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 [11], CompCert [12]) and translation validators [14] 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 [2], FCF [15]) 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 (as we do for EdDSA and X25519) and that the protocol is secure (out of scope for this paper). The work by Beringer et al. [3] 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 and Conclusion

Our compiler is already used by one very popular software project, but a number of improvements would help broaden its appeal. We would like to shrink our trusted base by connecting to a verified compiler targeting assembly. However, existing compilers are not smart enough at applying the constant-factor optimizations that are common in this domain. Thus another fruitful future-work area is studying those optimizations, principally combined register allocation and instruction scheduling, even independently of proof. Finally, we believe our general approach sketched in Section 2 ought to be a good fit for several other cryptographic domains, including hyper-elliptic-curve cryptography, RSA with a fixed modulus size, and lattice-based cryptography. Especially the last of these is undergoing an exciting period of protocol experimentation, making it especially valuable to provide an automatic compiler from high-level protocol descriptions to performance-competitive machine code.

References

[1] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. 2017. Jasmin: High-Assurance and High-Speed Cryptography. In *Proc. CCS*.

[2] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella-Béguelin. 2009. Formal Certification of Code-Based Cryptographic Proofs. In *36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009*. ACM, 90–101. http://software.imdea.org/~szanella/Zanella.2009.POPL.pdf

[3] Lennart Beringer, Adam Petcher, Katherine Q. Ye, and Andrew W. Appel. 2015. Verified Correctness and Security of OpenSSL HMAC. In *24th USENIX Security Symposium*. 207–221. https://www.usenix.org/system/files/conference/usenixsecurity15/sec15-paper-beringer.pdf

[4] Daniel J. Bernstein and Tanja Lange. 2017. eBACS: ECRYPT Benchmarking of Cryptographic Systems. https://bench.cr.yp.to/supercop/supercop-20170228.tar.xz

[5] Daniel J. Bernstein and Peter Schwabe. 2016. (2016). http://gfverif.cryptojedi.org/

[6] Daniel J. Bernstein, Bernard van Gastel, Wesley Janssen, Tanja Lange, Peter Schwabe, and Sjaak Smetsers. 2015. TweetNaCl: A crypto library in 100 tweets. In *Progress in Cryptology – LATINCRYPT 2014 (Lecture Notes in Computer Science)*, Diego Aranha and Alfred Menezes (Eds.), Vol. 8895. Springer-Verlag Berlin Heidelberg, 64–83. http://cryptojedi.org/papers/#tweetnacl Document ID: c74b5bbf605ba02ad8d9e49f04aca9a2.

[7] Barry Bond, Chris Hawblitzel, Manos Kapritsos, Rustan Leino, Jay Lorch, Bryan Parno, Ashay Rane, Srinath Setty, and Laure Thompson. 2017. Vale: Verifying High-Performance Cryptographic Assembly Code. In *Proc. USENIX Security*.

[8] Yu-Fang Chen, Chang-Hong Hsu, Hsin-Hung Lin, Peter Schwabe, Ming-Hsien Tsai, Bow-Yaw Wang, Bo-Yin Yang, and Shang-Yi Yang. 2014. Verifying Curve25519 Software. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, CCS'14*. ACM, 299–309. http://cryptojedi.org/papers/#verify25519 Document ID: 55ab8668ce87d857c02a5b2d56d7da38.

[9] Adam Chlipala. 2008. Parametric Higher-Order Abstract Syntax for Mechanized Semantics. In *ICFP'08: Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*. http://adam.chlipala.net/papers/PhoasICFP08/

[10] Matteo Frigo and Steven G. Johnson. 2005. The Design and Implementation of FFTW3. *Proc. IEEE* 93, 2 (2005), 216–231. http://www.fftw.org/fftw-paper-ieee.pdf Special issue on "Program Generation, Optimization, and Platform Adaptation".

[11] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: A Verified Implementation of ML. In *POPL '14: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, 179–191. https://cakeml.org/popl14.pdf

[12] Xavier Leroy. 2009. A Formally Verified Compiler Back-end. *J. Autom. Reason.* 43, 4 (Dec. 2009), 363–446. http://gallium.inria.fr/~xleroy/publi/compcert-backend.pdf

[13] Magnus O. Myreen and Gregorio Curello. 2013. A Verified Bignum Implementation in x86-64 Machine Code. In *Proc. CPP*. http://www.cse.chalmers.se/~myreen/cpp13.pdf

[14] George C. Necula. 2000. Translation Validation for an Optimizing Compiler. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI '00)*. ACM, New York, NY, USA, 83–94. https://people.eecs.berkeley.edu/~necula/Papers/tv_pldi00.pdf

[15] Adam Petcher and Greg Morrisett. 2015. The Foundational Cryptography Framework. In *Proceedings of the 4th International Conference on Principles of Security and Trust - Volume 9036*. Springer-Verlag New York, Inc., New York, NY, USA, 53–72. http://adam.petcher.net/papers/FCF.pdf

[16] Loïc Pottier. 2010. Connecting Gröbner Bases Programs with Coq to do Proofs in Algebra, Geometry and Arithmetics. *CoRR* abs/1007.3615 (2010). http://arxiv.org/abs/1007.3615

[17] Jonathan Protzenko, Jean-Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella-Béguelin, Antoine Delignat-Lavaud, Cătălin Hriţcu, Karthikeyan Bhargavan, Cédric Fournet, and Nikhil Swamy. 2017. Verified Low-level Programming Embedded in F*. *Proc. ACM Program. Lang.* 1, ICFP, Article 17 (Aug. 2017), 29 pages. https://doi.org/10.1145/3110261

[18] M. Püschel, B. Singer, J. Xiong, J. M. F. Moura, J. Johnson, D. Padua, M. M. Veloso, and R. W. Johnson. 2004. 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 (2004), 21–45. https://users.ece.cmu.edu/~moura/papers/pueschelmouraetal-highperfcomp-feb04.pdf

[19] Tiark Rompf and Martin Odersky. 2010. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled DSLs. *Proceedings of the GPCE* (2010). https://infoscience.epfl.ch/record/150347/files/gpce63-rompf.pdf

[20] Tim Sheard and Simon Peyton Jones. 2016. Template Meta-programming for Haskell. (2 2016). https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/meta-haskell.pdf orig. 2002.

[21] Jean Karim Zinzindohoue, Evmorfia-Iro Bartzia, and Karthikeyan Bhargavan. 2016. A Verified Extensible Library of Elliptic Curves. In *IEEE Computer Security Foundations Symposium (CSF)*.

[22] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. 2017. HACL*: A Verified Modern Cryptographic Library. In *Proc. CCS*.

A Study of Bugs in Similar Crypto Code

The custom code that the experts write often has serious correctness and security bugs. To get a sense of the details, we surveyed project bug trackers and other Internet sources, stopping after finding 27 bugs (each hyperlinked to its bug report) in implementation of nontrivial cryptography-specific optimizations. Fig. 4 summarizes our findings, in terms of 5 emergent categories. The first three categories have to do with representing large integers using multiple machine-word-sized integers, with custom positional number systems. Carrying is fairly analogous to the same concept in grade-school arithmetic, and canonicalization involves converting back from a custom representation into a standard one. Elliptic curve formulas are part of high-level algebraic optimizations, above the level of operations on single large integers. Crypto primitives bring it all together to provide functionality like digital signatures.

Here is a sampling of root causes behind bugs.

- Mechanical errors: One of the two bugs uncovered in OpenSSL issue 3607 was summarized by its author as "Got math wrong :-(", which we think referred to a pencil-and-paper execution of numerical range analysis. 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." In `ed25519-amd64-64-24k`, one of 16,184 repetitive (and handwritten) lines should have been `r2 += 0 + carry` instead of `r1 += 0 + carry` [6, p. 2].
- Confusion over intermediate specifications: OpenSSL bug 1953 was traced back to confusion between the postconditions of exact division with remainder and an operation 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 $10 \cdot 2^{-29}$.
- Mathematical misconceptions: The CryptoNote doublespending bug arose from use of an algorithm on a composite-order elliptic curve when it is only applicable in a prime-order group.

B Full Results of Many-Primes Experiments

Tables 2 and 3 contain the full results of our performance experiments on many primes. Recall the basic experimental setup:

- Scrape all prime numbers mentioned in the archives of the ECC mailing list at `moderncrypto.org`. Crucially, we record not just the numeric values of the primes but also the *ways in which they are expressed* in terms of additions and subtractions of powers of 2 and small multiples thereof.
- We run a small Python script (shorter than 300 lines) to inspect the shapes of these prime formulas, using simple heuristics to choose the parameters to our Coq library: not just a prime modulus of arithmetic but also how to divide a big integer into digits and which sequence of carry operations to perform in modular reduction. Actually, the script generates four variants, by considering 64-bit vs. 32-bit hardware architectures and by considering the Montgomery and Solinas arithmetic strategies. The main operation defined in each case is a Montgomery ladder step.
- We run our Coq pipeline on every variant, culminating in C code pretty-printed for each one.
- 64-bit configurations are compiled and run on an x86 Linux desktop machine, while 32-bit configurations are compiled and run on an ARM Android mobile device. We save the running time of each variation.
- We also compile and run fixed C and C++ implementations using `libgmp`.

The three comparison implementations are:

- GMP C constant-time, the best comparison with the goals of the code we generate, since running time is required to be independent of integer inputs
- GMP C variable time, taking advantage of additional optimizations that leak input values through timing
- GMP C++, the only one of the comparison implementations that does not include manual memory management

All three comparison programs are conventional in that they are fixed C or C++ programs, where the prime modulus is set as a preprocessor macro. It is up to GCC and `libgmp` to take advantage of properties of each modulus. The final column in each table shows how much better our specialized generation does. We take the ratio of variable-time C GMP (the fastest GMP code) to whichever of our generated variants is faster.

Some columns in the tables contain dashes in place of numbers of seconds needed for one trial. Those spots indicate configurations where our Coq compilation currently times out or exhausts available memory. Considering that Coq is not designed as a platform for executing an optimizing compiler, we are pleased that we get as many successful compilations as we do! However, we continue working on optimizations to our implementation, to push up the size of prime whose code we can compile quickly. The timing bottleneck is generally in reification, where repeated inefficient manipulation of terms and contexts by Ltac incurs significant overhead. The memory bottleneck generally shows up at `Qed`-time. Note also that some configurations are expected to fail to build, for instance when applying the Solinas strategy to so-called "Montgomery-friendly" primes like $2^{256} - 88 \cdot 2^{240} - 1$, where implementation experts would never choose Solinas.

Reference	Specification	Implementation	Defect
Carrying			
go#13515	Modular exponentiation	uintptr-sized Montgomery form, Go	carry handling
NaCl ed25519 (p. 2)	F25519 mul, square	64-bit pseudo-Mersenne, AMD64	carry handling
openssl#ef5c9b11	Modular exponentiation	64-bit Montgomery form, AMD64	carry handling
openssl#74acf42c	Poly1305	multiple implementations	carry handling
nettle#09e3ce4d	secp-256r1 modular reduction		carry handling
CVE-2017-3732	x^2 mod m	Montgomery form, AMD64 assembly	carry, exploitable
openssl#1593	P384 modular reduction	carry handling	carry, exploitable
tweetnacl-U32	irrelevant	bit-twiddly C	‘sizeof(long)!=32‘
Canonicalization			
donna#8edc799f	GF($2^{255} - 19$) internal to wire	32-bit pseudo-Mersenne, C	non-canonical
openssl#c2633b8f	a + b mod p256	Montgomery form, AMD64 assembly	non-canonical
tweetnacl-m15	GF($2^{255} - 19$) freeze	bit-twiddly C	bounds? typo?
Misc. number system			
openssl#3607	P256 field element squaring	64-bit Montgomery form, AMD64	limb overflow
openssl#0c687d7e	Poly1305	32-bit pseudo-Mersenne, x86 and ARM	bad truncation
CVE-2014-3570	Bignum squaring	asm	limb overflow
ic#237002094	Barrett reduction for p256	1 conditional subtraction instead of 2	no counterexample
go#fa09811d	poly1305 reduction	AMD64 asm, missing subtraction of 3	found quickly
openssl#a970db05	Poly1305	Lazy reduction in x86 asm	lost bit 59
openssl#6825d74b	Poly1305	AVX2 addition and reduction	bounds?
ed25519.py	Ed25519	accepts signatures other impls reject	missing h mod l
bitcoin#eed71d85	ECDSA-secp256k1 x*B	mixed addition Jacobian+Affine	missing case
Elliptic Curves			
openjdk#01781d7e	EC scalarmult	mixed addition Jacobian+Affine	missing case
jose-adobe	ECDH-ES	5 libraries	not on curve
invalid-curve	NIST ECDH	Irrelevant	not on curve
end-to-end#340	Curve25519 library	twisted Edwards coordinates	$(0, 1) = \infty$
openssl#59dfcabf	Weier. affine <-> Jacobian	Montgomery form, AMD64 and C	∞ confusion
Crypto Primitives			
socat#7	DH in Z*p	irrelevant	non-prime p
CVE-2006-4339	RSA-PKCS-1 sig. verification	irrelevant	padding check
CryptoNote	Anti-double-spending tag	additive curve25519 curve point	missed order(P) $\neq l$

Figure 4. Survey of bugs in algebra-based cryptography implementations

Among successful compilations, time ranges between tens of seconds and levels best run overnight.

C Weighing Our Code Base

We can also give a short quantitative summary of our code base, as a proxy for work required to develop and maintain it. Just the code for unsaturated arithmetic, described in Section 4, requires 160 lines (each 80 characters or less) of code and proof, which can then trivially be used to generate all unsaturated reduction examples in this paper. However, trying to synthesize a chained carry operation using this code would result in exponential blow-up due to loss of sharing. Rewriting functions in continuation-passing style to force sharing of subexpressions (as demonstrated in Section 2.4), the same library grows to around 1000 lines. The extensions for saturated arithmetic add 800 lines, and all proofs and code we needed to add for Montgomery reduction total 1500 lines. For comparison, the `amd64-51` implementation of X25519 contains 1900 lines of assembly code (with P256 implementations being substantially longer), and the translator from assembly to SMT-solver formulas used by Chen et al. [8] is 8800 lines of OCaml. The elliptic-curves library is rather thin: a total of 1300 lines of code for 3 curve shapes and 8 point formats, including all equivalence proofs. Certified-compiler-phase implementations are rather verbose and sometimes include more lines for parameters than actual code: the largest one is bounds inference (about 2500 lines code+proof), adding up to a total around 15,000 lines. With another 15,000 lines of utility lemmas and tactics that seem reasonable candidates to move into Coq's standard library, the total development adds up to 38,000 lines.

Operation	Handwritten	Generated	Ratio
Key generation	10965	10808	.98
Sign	10841	10807	.99
Verify signature	3056	2919	.95
Base-point ×	11177	11061	.98
Arbitrary-point ×	3552	3530	.99

Table 1. Performance details for Curve25519 generated code

D Performance in BoringSSL

Table 1 has more performance detail on our generated code (for Curve25519) integrated into BoringSSL, the cryptography library behind Google Chrome. We compare the version with our code with the one immediately before it, relying on handwritten code instead. Each measurement is a number of operations per second, taken as the median of three trials on an Intel Xeon with AVX2.

Prime	Our Code		GMP Code			Speed-up
	Sol.	Mont.	const time	var time	C++	
$2^{127}-1$	0.03	0.04	0.26	0.15	0.67	5.0
$2^{129}-25$	0.03	0.07	0.38	0.27	0.8	9.0
$2^{130}-5$	0.03	0.09	0.39	0.28	0.79	9.33
$2^{137}-13$	0.03	0.08	0.37	0.27	0.8	9.0
$2^{140}-27$	0.03	0.08	0.38	0.27	0.8	9.0
$2^{141}-9$	0.03	0.08	0.39	0.27	0.83	9.0
$2^{150}-3$	0.03	0.08	0.38	0.3	0.8	10.0
$2^{150}-5$	0.03	0.08	0.39	0.29	0.84	9.67
$2^{152}-17$	0.03	0.08	0.38	0.27	0.82	9.0
$2^{158}-15$	0.03	0.08	0.37	0.27	0.76	9.0
$2^{165}-25$	0.03	0.08	0.38	0.27	0.78	9.0
$2^{166}-5$	0.03	0.08	0.39	0.27	0.79	9.0
$2^{171}-19$	0.03	0.08	0.38	0.27	0.79	9.0
$2^{174}-17$	0.03	0.08	0.38	0.28	0.78	9.33
$2^{174}-3$	0.03	0.08	0.38	0.27	0.78	9.0
$2^{189}-25$	0.04	0.08	0.39	0.27	0.8	6.75
$2^{190}-11$	0.04	0.08	0.38	0.27	0.78	6.75
$2^{191}-19$	0.04	0.09	0.36	0.26	0.78	6.5
$2^{192}-2^{64}-1$	0.05	0.07	0.31	0.24	0.79	4.8
$2^{194}-33$	0.04	0.12	0.5	0.34	0.93	8.5
$2^{196}-15$	0.04	0.12	0.5	0.34	0.89	8.5
$2^{198}-17$	0.04	0.12	0.51	0.34	0.87	8.5
$2^{205}-45\cdot 2^{198}-1$	-	0.14	0.51	0.34	0.87	2.43
$2^{206}-5$	0.04	0.14	0.5	0.34	0.84	8.5
$2^{212}-29$	0.05	0.12	0.49	0.35	0.87	7.0
$2^{213}-3$	0.04	0.13	0.49	0.37	0.88	9.25
$2^{216}-2^{108}-1$	0.04	0.12	0.51	0.35	0.88	8.75
$2^{221}-3$	0.05	0.15	0.51	0.36	0.89	7.2
$2^{222}-117$	0.05	0.12	0.53	0.35	0.91	7.0
$2^{224}-2^{96}+1$	-	0.13	0.5	0.35	0.88	2.69
$2^{226}-5$	0.04	0.13	0.5	0.35	0.92	8.75
$2^{230}-27$	0.05	0.13	0.54	0.35	0.91	7.0
$2^{235}-15$	0.06	0.13	0.5	0.34	0.89	5.67
$2^{243}-9$	0.06	0.13	0.5	0.34	0.89	5.67
$2^{251}-9$	0.06	0.13	0.5	0.35	0.94	5.83
$2^{254}-127\cdot 2^{240}-1$	-	0.12	0.5	0.35	0.92	2.92
$2^{255}-19$	0.06	0.13	0.48	0.35	0.9	5.83
$2^{255}-765$	0.06	0.13	0.52	0.34	0.9	5.67
$2^{256}-189$	0.06	0.14	0.38	0.34	0.87	5.67
$2^{256}-2^{224}+2^{192}+2^{96}-1$	-	0.11	0.38	0.33	0.84	3.0
$2^{256}-2^{32}-977$	0.1	0.12	0.38	0.34	0.87	3.4
$2^{256}-4294968273$	0.14	0.13	0.37	0.34	0.86	2.62
$2^{256}-88\cdot 2^{240}-1$	-	0.11	0.39	0.34	0.88	3.09
$2^{266}-3$	0.06	0.18	0.66	0.45	1.13	7.5
$2^{285}-9$	0.06	0.18	0.73	0.43	0.97	7.17
$2^{291}-19$	0.07	0.18	0.68	0.42	1.0	6.0
$2^{321}-9$	0.1	0.26	0.8	0.54	1.18	5.4
$2^{322}-2^{161}-1$	0.07	0.27	0.83	0.53	1.15	7.57
$2^{336}-17$	0.1	0.27	0.8	0.53	1.11	5.3
$2^{336}-3$	0.09	0.27	0.86	0.53	1.08	5.89
$2^{338}-15$	0.1	0.25	0.8	0.54	1.06	5.4
$2^{369}-25$	0.13	0.26	0.79	0.52	1.1	4.0
$2^{379}-19$	0.12	0.26	0.79	0.55	1.07	4.58
$2^{382}-105$	0.13	0.25	0.92	0.57	1.11	4.38
$2^{383}-187$	0.13	0.28	0.75	0.5	1.05	3.85
$2^{383}-31$	0.13	0.26	0.75	0.51	1.05	3.92
$2^{383}-421$	0.13	0.25	0.76	0.51	1.06	3.92
$2^{384}-2^{128}-2^{96}+2^{32}-1$	-	0.25	0.64	0.47	0.98	1.88
$2^{384}-317$	0.13	0.26	0.67	0.48	1.0	3.69
$2^{384}-5\cdot 2^{368}-1$	-	0.23	0.63	0.46	0.99	2.0
$2^{384}-79\cdot 2^{376}-1$	-	0.23	0.62	0.46	0.99	2.0
$2^{389}-21$	0.13	-	0.97	0.6	1.22	4.62
$2^{401}-31$	0.14	-	0.97	0.61	1.17	4.36
$2^{413}-21$	0.16	-	0.99	0.62	1.22	3.88
$2^{414}-17$	0.15	-	0.98	0.6	1.21	4.0
$2^{416}-2^{208}-1$	0.14	-	0.98	0.6	1.16	4.29
$2^{444}-17$	0.17	-	0.96	0.6	1.2	3.53
$2^{448}-2^{224}-1$	0.12	-	0.79	0.52	1.06	4.33
$2^{450}-2^{225}-1$	0.13	-	1.22	0.74	1.34	5.69
$2^{452}-3$	0.16	-	1.24	0.71	1.32	4.44
$2^{468}-17$	0.16	-	1.23	0.71	1.29	4.44
$2^{480}-2^{240}-1$	0.13	-	1.18	0.71	1.28	5.46
$2^{488}-17$	0.19	-	1.2	0.7	1.28	3.68
$2^{489}-21$	0.2	-	1.17	0.69	1.27	3.45
$2^{495}-31$	0.19	-	1.17	0.69	1.3	3.63
$2^{510}-290\cdot 2^{496}-1$	-	-	1.2	0.7	1.28	-
$2^{511}-187$	0.25	-	1.13	0.66	1.21	2.64
$2^{511}-481$	0.25	-	1.12	0.66	1.24	2.64
$2^{512}-491\cdot 2^{496}-1$	-	-	0.99	0.62	1.15	-
$2^{512}-569$	0.24	-	0.95	0.62	1.14	2.58
$2^{521}-1$	0.18	-	1.4	0.81	1.44	4.5

Table 2. Full 64-bit benchmark data. Our code tried both Solinas and Montgomery implementations for each prime, and we test against three GMP-based implementations: one that is constant-time (gmpsec), one that is variable time (gmpvar), and GMP's C++ API. Our code is constant-time, so gmpsec is the best comparison; however, even with that constraint removed from GMP and not us, we compare favorably to gmpvar.

Prime	Our Code		GMP Code		Speedup
	Solinas	Mont.	const time	var time	
$2^{127} - 1$	0.3	1.19	2.86	3.23	9.53
$2^{129} - 25$	0.35	1.7	3.38	3.77	9.66
$2^{130} - 5$	0.44	1.87	3.56	3.79	8.09
$2^{137} - 13$	0.48	2.06	3.41	3.78	7.1
$2^{140} - 27$	0.51	1.98	3.43	3.77	6.73
$2^{141} - 9$	0.51	2.0	3.43	3.81	6.73
$2^{150} - 3$	0.42	2.0	3.56	3.79	8.48
$2^{150} - 5$	0.49	1.99	3.38	3.8	6.9
$2^{152} - 17$	0.5	1.96	3.4	3.82	6.8
$2^{158} - 15$	0.52	2.04	3.4	3.77	6.54
$2^{165} - 25$	0.59	2.46	4.02	4.45	6.81
$2^{166} - 5$	0.61	2.43	4.02	4.43	6.59
$2^{171} - 19$	0.57	2.68	4.04	4.51	7.09
$2^{174} - 17$	0.58	2.63	4.03	4.39	6.95
$2^{174} - 3$	0.61	2.62	4.02	4.4	6.59
$2^{189} - 25$	0.7	2.65	4.05	4.4	5.79
$2^{190} - 11$	0.71	2.64	4.1	4.42	5.77
$2^{191} - 19$	0.66	2.69	4.03	4.4	6.11
$2^{192} - 2^{64} - 1$	-	2.41	3.56	4.23	1.48
$2^{194} - 33$	0.75	-	4.66	4.94	6.21
$2^{196} - 15$	0.77	-	4.64	4.94	6.03
$2^{198} - 17$	0.76	-	4.72	4.97	6.21
$2^{205} - 45 \cdot 2^{198} - 1$	-	-	4.66	5.03	-
$2^{206} - 5$	0.76	-	4.62	4.91	6.08
$2^{212} - 29$	0.86	-	4.68	4.91	5.44
$2^{213} - 3$	0.7	-	4.68	4.94	6.69
$2^{216} - 2^{108} - 1$	0.7	-	4.67	4.92	6.67
$2^{221} - 3$	0.8	-	4.68	4.92	5.85
$2^{222} - 117$	0.87	-	4.72	4.87	5.43
$2^{224} - 2^{96} + 1$	-	-	4.13	4.85	-
$2^{226} - 5$	0.87	-	5.25	5.65	6.03
$2^{230} - 27$	0.83	-	5.29	5.71	6.37
$2^{235} - 15$	0.9	-	5.31	5.69	5.9
$2^{243} - 9$	0.86	-	5.29	5.62	6.15
$2^{251} - 9$	1.12	-	5.3	5.65	4.73
$2^{254} - 127 \cdot 2^{240} - 1$	-	3.97	5.26	5.7	1.32
$2^{255} - 19$	1.01	-	5.25	5.7	5.2
$2^{255} - 765$	1.43	-	5.27	5.71	3.69
$2^{256} - 189$	1.2	-	4.71	5.49	3.93
$2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$	-	-	4.7	5.46	-
$2^{256} - 2^{32} - 977$	1.65	-	4.72	5.45	2.86
$2^{256} - 4294968273$	-	-	4.77	5.48	-
$2^{256} - 88 \cdot 2^{240} - 1$	-	-	4.78	5.46	-
$2^{266} - 3$	1.01	-	6.1	6.32	6.04
$2^{285} - 9$	1.13	-	6.13	6.34	5.42
$2^{291} - 19$	1.33	-	6.94	6.98	5.22
$2^{321} - 9$	1.72	-	7.6	7.66	4.42
$2^{322} - 2^{161} - 1$	1.37	-	7.66	7.74	5.59
$2^{336} - 17$	1.67	-	7.64	7.74	4.57
$2^{336} - 3$	1.59	-	7.58	7.69	4.77
$2^{338} - 15$	1.7	-	7.66	7.67	4.51
$2^{369} - 25$	2.44	-	8.41	9.03	3.45
$2^{379} - 19$	2.47	-	8.44	9.25	3.42
$2^{382} - 105$	2.66	-	8.41	9.04	3.16
$2^{383} - 187$	2.63	-	8.44	9.11	3.21
$2^{383} - 31$	2.6	-	8.47	9.13	3.26
$2^{383} - 421$	3.58	-	8.45	9.11	2.36
$2^{384} - 2^{128} - 2^{96} + 2^{32} - 1$	-	-	7.62	8.8	-
$2^{384} - 317$	3.95	-	7.62	8.82	1.93
$2^{384} - 5 \cdot 2^{368} - 1$	-	-	7.64	8.94	-
$2^{384} - 79 \cdot 2^{376} - 1$	-	-	7.66	8.84	-
$2^{389} - 21$	2.89	-	9.41	9.93	3.26
$2^{401} - 31$	2.85	-	9.35	9.92	3.28
$2^{413} - 21$	3.53	-	9.48	9.93	2.69
$2^{414} - 17$	3.72	-	9.4	9.86	2.53
$2^{416} - 2^{208} - 1$	2.48	-	8.54	9.67	3.44
$2^{444} - 17$	3.7	-	10.31	10.89	2.79
$2^{448} - 2^{224} - 1$	3.18	-	9.57	10.51	3.01
$2^{450} - 2^{225} - 1$	-	-	11.37	11.63	-
$2^{452} - 3$	3.23	-	11.33	11.63	3.51
$2^{468} - 17$	3.2	-	11.37	11.63	3.55
$2^{480} - 2^{240} - 1$	3.58	-	10.47	11.33	2.92
$2^{488} - 17$	7.99	-	12.23	12.92	1.53
$2^{489} - 21$	7.7	-	12.26	12.81	1.59
$2^{495} - 31$	6.07	-	12.2	13.1	2.01
$2^{510} - 290 \cdot 2^{496} - 1$	-	-	12.17	12.9	-
$2^{511} - 187$	9.73	-	12.21	13.07	1.25
$2^{511} - 481$	-	-	12.23	12.9	-
$2^{512} - 491 \cdot 2^{496} - 1$	-	-	11.26	12.58	-
$2^{512} - 569$	-	-	11.23	12.55	-
$2^{521} - 1$	3.9	-	13.3	13.91	3.41

Table 3. Full 32-bit benchmark data. Many of the 32-bit Montgomery implementations exceeded the one-hour timeout for proofs, because 32-bit code involves approximately twice as many operations. The C++ GMP program was not benchmarked on 32-bit.