# Simple High-Level Code For Cryptographic Arithmetic – With Proofs, Without Compromises

Andres Erbsen Jade Philipoom Jason Gross Robert Sloan Adam Chlipala
MIT CSAIL,
Cambridge, MA, USA
{andreser, jadep, jgross}@mit.edu, rob.sloan@alum.mit.edu, adamc@csail.mit.edu

***Abstract*—We introduce a new approach for implementing cryptographic arithmetic in short high-level code with machine-checked proofs of functional correctness. We further demonstrate that simple partial evaluation is sufficient to transform into the fastest-known C code, breaking the decades-old pattern that the only fast implementations are those whose instruction-level steps were written out by hand.**

**These techniques were used to build an elliptic-curve library that achieves competitive performance for 80 prime fields and multiple CPU architectures, showing that implementation and proof effort scales with the number and complexity of conceptually different algorithms, not their use cases. As one outcome, we present the first verified high-performance implementation of P-256, the most widely used elliptic curve. Implementations from our library were included in BoringSSL to replace existing specialized code, for inclusion in several large deployments for Chrome, Android, and CloudFlare.**

## I. Motivation

The predominant practice today is to implement cryptographic primitives at the lowest possible level of abstraction: usually flat C code, ideally assembly language if engineering time is plentiful. Low-level implementation has been the only way to achieve good performance, ensure that execution time does not leak secret information, and enable the code to be called from any programming language. Yet placing the burden of every detail on the programmer also leaves a lot to be desired. The demand for implementation experts' time is high, resulting in an ever-growing backlog of algorithms to be reimplemented in specialized code "as soon as possible." Worse, bugs are also common enough that it is often not economical to go beyond fixing the behavior for problematic inputs and understand the root cause and impact of the defect.

There is hope that these issues can eventually be overcome by redoubling community efforts to produce high-quality crypto code. However, even the most renowned implementors are not infallible, and if a typo among thousands of lines of monotonous code can only be recognized as incorrect by top experts,[1] it is reasonable to expect diminishing returns for achieving correctness through manual review. A number of tools for computer-aided verification have been proposed [1]–[6], enabling developers to eliminate incorrect-output bugs conclusively. Three solid implementations of the X25519 Diffie-Hellman function were verified. In the beautiful world where X25519 was the only arithmetic-based crypto primitive we need, now would be the time to declare victory and go home. Yet most of the Internet still uses P-256, and the current proposals for post-quantum cryptosystems are far from Curve25519's combination of performance and simplicity.

A closer look into the development and verification process (with an eye towards replicating it for another function) reveals a discouraging amount of duplicated effort. First, expert time is spent on mechanically writing out the instruction-level steps required to perform the desired arithmetic, drawing from experience and folklore. Then, expert time is spent to optimize the code by applying a carefully chosen combination of simple transformations. Finally, expert time is spent on annotating the now-munged code with assertions relating values of run-time variables to specification variables, until the SMT solver (used as a black box) is convinced that each assertion is implied by the previous ones. As the number of steps required to multiply two numbers is quadratic in the size of those numbers, effort grows accordingly: for example "Verifying Curve25519 Software" [1] reports that multiplication modulo $2^{255} - 19$ required 27 times the number of assertions used for $2^{127} - 1$. It is important to note that both examples use *the same* simple modular multiplication algorithm, and the same correctness argument should apply, just with different parameters. This is just one example of how manual repetition of algorithmic steps inside primitive cryptographic arithmetic implementations makes them notoriously difficult to write, review, or prove correct.

Hence we suggest a rather different way of implementing cryptographic primitives: to implement general algorithms in the style that best illustrates their correctness arguments and then derive parameter-specific low-level code through partial evaluation. For example, the algorithm used for multiplication modulo $2^{127} - 1$ and $2^{255} - 19$ is first encoded as a purely functional program operating on lists of digit-weight pairs: it generates a list of partial products, shifts down the terms with weight greater than the modulus, and scales them accordingly. It is then instantiated with lists of concrete weights and length, for example ten limbs with weights $2^{\lceil 25.5i \rceil}$, and partially evaluated so that no run-time manipulation of lists and weights remains, resulting in straight-line code very similar to what an expert would have written. As the optimization pipeline is proven correct once and for all, correctness of all code generated from a high-level algorithm can be proven directly

[1]We refer to the `ed25519-amd64-64-24k` bug that is discussed along with other instructive real-world implementation defects in Appendix A.

on its high-level implementation. A nice side benefit of this approach is that the same high-level algorithm and proof can be used for multiple target configurations with very little effort, enabling the use of fast implementations in cases that were previously not worth the effort.

In practice, we use the fast arithmetic implementations reported on here in conjunction with a library of verified cryptographic algorithms, the rest of which deviates much less from common implementation practice. All formal reasoning is done in the Coq proof assistant, and the overall trusted computing base also includes a simple pretty-printer and the C language toolchain. The development, called **Fiat Cryptography**, is available under the MIT license at:

`https://github.com/mit-plv/fiat-crypto`

The approach we advocate for is already used in widely deployed code. Most applications relying on the implementations described in this paper are using them through BoringSSL, Google's OpenSSL-derived crypto library that includes our Curve25519 and P-256 implementations. A high-profile user is the Chrome Web browser, so today about half of HTTPS connections opened by Web browsers worldwide use our fast verified code (Chrome versions since 65 have about 60% market share [7], and 90% of connections use X25519 or P-256 [8]). BoringSSL is also used in Android and on a majority of Google's servers, so a Google search from Chrome would likely be relying on our code on both sides of the connection. Maintainers appreciate the reduced need to worry about bugs.

## II. Overview

We will explain how we write algorithm implementations in a high-level way that keeps the structural principles as clear and readable as possible, proving functional correctness. Our explanation builds up to showing how other security constraints can be satisfied without sacrificing the appeal of our new strategy. Our output code follows industry-standard "constant-time" coding practices and is only bested in throughput by platform-specific assembly-language implementations (and only by a modest margin). This section will give a tour of the essential ingredients of our methodology, and the next section will explain in detail how to implement field arithmetic for implementation-friendly primes like $2^{255} - 19$. Building on this intuition (and code!), Section III-F culminates in a specializable implementation of the word-by-word Montgomery reduction algorithm for multiplication modulo any prime. Section IV will explain the details of the compiler that turns the described implementations into the kind of code that implementation experts write by hand for each arithmetic primitive. We report performance measurements in Section V and discuss related work and future directions in Sections VI and VII.

Fig. 1 gives a concrete example of what our framework provides. The algorithm in the top half of the figure is a transcription of the word-by-word Montgomery reduction algorithm, following Gueron's Algorithm 4 [9]. As is required by the current implementation of our optimization pipeline, this parameter-agnostic code is written in *continuation-passing style* where functions do not return normally but instead take as arguments callbacks to invoke when answers become available. The notation `λ '(x, y), ...` encodes an anonymous function with arguments `x` and `y`.

**Input:**

```
Definition wwmm_step A B k S ret :=
  divmod_r_cps A (λ '(A, T1),
  @scmul_cps r _ T1 B _ (λ aB,
  @add_cps r _ S aB _ (λ S,
  divmod_r_cps S (λ '(_, s),
  mul_split_cps' r s k (λ '(q, _),
  @scmul_cps r _ q M _ (λ qM,
  @add_longer_cps r _ S qM _ (λ S,
  divmod_r_cps S (λ '(S, _),
  @drop_high_cps (S R_numlimbs) S _ (λ S,
  ret (A, S) ))))))))).

Fixpoint wwmm_loop A B k len_A ret :=
  match len_A with
  | O => ret
  | S len_A' => λ '(A, S),
    wwmm_step A B k S (wwmm_loop A B k len_A' ret)
  end.
```

**Output:**

```
void wwmm_p256(u64 out[4], u64 x[4], u64 y[4]) {
  u64 x17, x18 = mulx_u64(x[0], y[0]);
  u64 x20, x21 = mulx_u64(x[0], y[1]);
  u64 x23, x24 = mulx_u64(x[0], y[2]);
  u64 x26, x27 = mulx_u64(x[0], y[3]);
  u64 x29, u8 x30 = addcarryx_u64(0x0, x18, x20);
  u64 x32, u8 x33 = addcarryx_u64(x30, x21, x23);
  u64 x35, u8 x36 = addcarryx_u64(x33, x24, x26);
  u64 x38, u8 _ = addcarryx_u64(0x0, x36, x27);
  u64 x41, x42 = mulx_u64(x17, 0xffffffffffffffff);
  u64 x44, x45 = mulx_u64(x17, 0xffffffff);
  u64 x47, x48 = mulx_u64(x17, 0xffffffff00000001);
  u64 x50, u8 x51 = addcarryx_u64(0x0, x42, x44);
  // 100 more lines...
```

Fig. 1: Word-by-word Montgomery Multiplication, output specialized to P-256 ($2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$)

The bottom half of the figure shows the result of specializing the general algorithm to P-256, the elliptic curve most commonly used in TLS and unfortunately not supported by any of the verified libraries discussed earlier. Note that some operations influence the generated code by creating explicit instruction sequences (highlighted in matching colors), while other operations instead control which input expressions should be fed into others. The number of low-level lines per high-level line varies between curves and target hardware architectures. The generated code is accompanied by a Coq proof of correctness stated in terms of the straightline-code language semantics and the Coq standard-library definitions of integer arithmetic.

### A. High-Level Arithmetic Library Designed For Specialization

At the most basic level, all cryptographic arithmetic algorithms come down to implementing large mathematical objects (integers or polynomials of several hundred bits) in terms of arithmetic provided by the hardware platform (for example integers modulo $2^{64}$). To start out with, let us consider the example of addition, with the simplifying precondition that no digits exceed half of the hardware-supported maximum.

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

To give the specification of this function acting on a base-$2^{64}$ little-endian representation, we define an abstraction function evaluating each digit list $\ell$ into a single number.

$$\begin{aligned}&\lfloor \_ \rfloor : \textbf{num} \to \mathbb{N}\\ &\lfloor a :: as \rfloor = a + 2^{64} \lfloor as \rfloor\\ &\lfloor [] \rfloor = 0\end{aligned}$$

To prove correctness, we show (by naive induction on $a$) that evaluating the result of **add** behaves as expected:

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

Using the standard list type in Coq, the theorem becomes

```
Lemma eval_add : ∀a b, ⌊add a b⌋ = ⌊a⌋+⌊b⌋.
 induction a,b;cbn; rewrite ?IHa;nia. Qed.
```

The above proof is only possible because we are intentionally avoiding details of the underlying machine arithmetic – if the list of digits were defined to contain 64-bit words instead of $\mathbb{N}$, repeated application of **add** would eventually result in integer overflow. This simplification is an 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 postponing this reasoning: generally we care about the *context* of higher-level code calling our arithmetic primitives. In an arbitrary context, an **add** implemented using 64-bit words would need to propagate carries from each word to the next, causing an unnecessary slowdown when called with inputs known to be small.

### B. Partial Evaluation For Specific Parameters

It is impossible to achieve competitive performance with arithmetic code that manipulates dynamically allocated lists at runtime. The fastest code will implement, for example, 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 that generalize the patterns lurking behind hand-written implementations favored today.

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 **add** $[a_1, a_2, a_3]\ [b_1, b_2, b_3]$, where the $a_i$s and $b_i$s 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 $\lambda$-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 and $+$ unreduced but apply other rules.

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

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. To persuade Coq's built-in term reduction to do what we want, we first translate arithmetic operations to *continuation-passing style*. Concretely, we can rewrite **add**.

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

Reduction turns this function into assembly-like code.

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

When **add′** is applied to a particular continuation, we can also reduce away the result list. Chaining together sequences of function calls leads to idiomatic and efficient assembly-style code, based just on Coq's normal term reduction, preserving sharing of let-bound variables. This level of inlining is common for the inner loops of crypto primitives, and it will also simplify the static analysis described in the next subsection.

### C. Word-Size 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, but so far it is only proven correct for natural numbers. 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.

The bounds-inference pass works by standard abstract interpretation with intervals. As inputs, we require lower and upper bounds for the integer values of all arguments of a function. These bounds are then pushed through all operations to infer bounds for temporary variables. Each temporary is assigned the smallest bit width that can accommodate its full interval.

| prime | architecture | # limbs | base | representation (distributing large $x$ into $x_0...x_n$) |
|---|---|---|---|---|
| $2^{256}-2^{224}+2^{192}+2^{96}-1$ (P-256) | 64-bit | 4 | $2^{64}$ | $x = x_0 + 2^{64}x_1 + 2^{128}x_2 + 2^{192}x_3$ |
| $2^{255}-19$ (Curve25519) | 64-bit | 5 | $2^{51}$ | $x = x_0 + 2^{51}x_1 + 2^{102}x_2 + 2^{153}x_3 + 2^{204}x_4$ |
| $2^{255}-19$ (Curve25519) | 32-bit | 10 | $2^{25.5}$ | $x = x_0 + 2^{26}x_1 + 2^{51}x_2 + 2^{77}x_3 + ... + 2^{204}x_8 + 2^{230}x_9$ |
| $2^{448}-2^{224}-1$ (p448) | 64-bit | 8 | $2^{56}$ | $x = x_0 + 2^{56}x_1 + 2^{112}x_2 + ... + 2^{392}x_7$ |
| $2^{127}-1$ | 64-bit | 3 | $2^{42.5}$ | $x = x_0 + 2^{43}x_1 + 2^{85}x_2$ |

Fig. 2: Examples of big-integer representations for different primes and integer widths

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.

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

Note how we may infer different temporary widths based on different bounds for arguments. 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.

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 IV explains in more detail.

### D. *Compilation To Constant-Time Machine Code*

What results is straightline code very similar to that written by hand by experts, represented as ASTs in a simple language with arithmetic and bitwise operators. Our correctness proofs connect this AST to specifications in terms of integer arithmetic, such as the one for **add** above. All operations provided in our lowest-level AST are implemented with input-independent execution time in many commodity compilers and processors, and if so, our generated code is trivially free of timing leaks. Each function is pretty-printed as C code and compiled with a normal C compiler, ready to be benchmarked or included in a library. We are well aware that top implementation experts can translate C to assembly better than the compilers, and we do not try to compete with them: while better instruction scheduling and register allocation for arithmetic-heavy code would definitely be valuable, it is outside the scope of this project. Nevertheless, we are excited to report that our library generates the fastest-known C code for all operations we have benchmarked (Section V).

## III. Arithmetic Template Library

Recall Section II-A's toy **add** function for little-endian big integers. We will now describe our full-scale library, starting with the core *unsaturated* arithmetic subset, the foundation for all big-integer arithmetic in our development. For those who prefer to read code, we suggest `src/Demo.v` in the framework's source code, which contains a succinct standalone development of the unsaturated-arithmetic library up to and including prime-shape-aware modular reduction. The concrete examples derived in this section are within established implementation practice, and an expert would be able to reproduce them given an informal description of the strategy. Our contribution is to encode this general wisdom in concrete algorithms and provide these with correctness certificates without sacrificing the programmer's sanity.

### A. *Multi-Limbed Arithmetic*

Cryptographic modular arithmetic implementations distribute very large numbers across smaller "limbs" of 32- or 64-bit integers. Fig. 2 shows a small sample of fast representations for different primes. Notice that many of these implementations use bases other than $2^{32}$ or $2^{64}$, leaving bits unused in each limb: these are called *unsaturated* representations. Conversely, the ones using all available bits are called *saturated*.

Another interesting feature shown in the examples is that the exponents of some bases, such as the original 32-bit Curve25519 representation [10], are not whole numbers. In the actual representation, this choice corresponds to an alternating pattern, so "base $2^{25.5}$" uses 26 bits in the first limb, 25 in the second, 26 in the third, and so on. Because of the alternation, these are called *mixed-radix* bases, as opposed to *uniform-radix* ones.

These unorthodox big integers are fast primarily because of a specific modular-reduction trick, which is most efficient when the number of bits in the prime corresponds to a limb boundary. For instance, reduction modulo $2^{255} - 19$ is fastest when the bitwidths of a prefix of limbs sum to 255. Every unsaturated representation in our examples is designed to fulfill this criterion.

### B. *Partial Modular Reduction*

Suppose we are working modulo a $k$-bit prime $m$. Multiplying two $k$-bit numbers can produce up to $2k$ bits, potentially much larger than $m$. However, if we only care about what the result is mod $m$, we can perform a *partial modular*

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

Fig. 3: Distributing terms for multiplication mod $2^{127} - 1$

*reduction* to reduce the upper bound while preserving modular equivalence. (The reduction is "partial" because the result is not guaranteed to be the minimal residue.)

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 $c_i$ facilitate fast arithmetic.

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 *split* $x$ by finding $a$ and $b$ such that $x = a + sb$. Then, a simple derivation yields a division-free procedure for partial modular reduction.

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

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 $a = x$ and $b = 0$ would make this formula a no-op. One might pick $a = x \bmod s$, although the formula does not require it. Here is where careful choices of big-integer representation help: having $s$ be at a limb boundary allows for a good split without any computation!

Our Coq proof of this trick is reproduced here:

```
Lemma reduction_rule a b s c (_: s-c <> 0)
 : (a + s * b) mod (s - c) =
   (a + c * b) mod (s - c).
Proof.
 replace (a+s*b) with ((a+c*b)+b*(s-c)).
 rewrite add_mod,mod_mult,add_0_r,mod_mod.
 all: auto; nsatz.
Qed.
```

## C. Example: Multiplication Modulo $2^{127} - 1$

Before describing the general modular-reduction algorithm implemented in our library, we will walk through multiplication specialized to just one modulus and representation. To simplify matters a bit, we use the (relatively) small 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 Fig. 3.

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, the last two digits can be rearranged so that the modular-reduction rule applies:

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

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 specifying the limbs of a single-width answer:

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

## D. Associational Representation

As is evident by now, the most efficient code makes use of sophisticated and specific big-number representations, but many concrete implementations operate on the same set of underlying principles. Our strategy of writing high-level templates allows us to capture the underlying principles in a fully general way, without committing to a run-time representation. Abstracting away the implementation-specific details, like the exact number of limbs or whether the base system is mixed- or uniform-radix, enables simple high-level proofs of all necessary properties.

A key insight that allows us to simplify the arithmetic proofs is to use multiple different representations for big integers, as long as changes between these representations can be simplified away once the prime shape is known. We have two main representations of big integers in the core library: *associational* and *positional*. A big integer in associational form is represented as a list of pairs; the first element in each pair is a *weight*, known at compile time, and the second is a runtime value. The decimal number 95 might be encoded as `[(16, 5); (1, 15)]`, or equivalently as `[(1, 5); (10, 9)]` or `[(1, 1); (1, 6); (44, 2)]`. As long as the sum of products is correct, we do not care about the order, whether the weights are powers of each other, or whether there are multiple pairs of equal weight.

```
Definition mul (p q:list(Z*Z)) : list(Z*Z)
:= flat_map (λt,
    map (λt',(fst t*fst t', snd t*snd t'))
   q) p.

Lemma eval_map_mul (a x:Z) (p:list (Z*Z))
  : eval (map (λt, (a*fst t, x*snd t)) p)
  = a*x * eval p.
Proof. induction p; push; nsatz.  Qed.

Lemma eval_mul p q
  : eval (mul p q) = eval p * eval q.
Proof.
induction p; cbv [mul]; push; nsatz. Qed.
```

Fig. 4: Definition and correctness proof of multiplication

In associational format, arithmetic operations are extremely easy to reason about. Addition is just concatenation of the lists. Schoolbook multiplication (see Fig. 4) is also simple: $(a_1 \cdot x_1 + \ldots)(b_1 \cdot y_1 + \ldots) = (a_1 b_1 \cdot x_1 y_1 + \ldots)$, where $a_1 b_1$ is the new compile-time-known weight. Because of the flexibility allowed by associational representation, we do not have to worry about whether the new compile-time weight produced is actually in the base. If we are working with the base $2^{25.5}$ in associational representation, and we multiply two pairs that both have weight $2^{26}$, we can just keep the $2^{52}$ rather than the base's $2^{51}$ weight.

Positional representation, on the other hand, uses lists of runtime values with weight functions. There is only one runtime value for each weight, and order matters. This format is closer to how a number would actually be represented at runtime. Separating associational from positional representations lets us separate reasoning about arithmetic from reasoning about making weights line up. Especially in a system that must account for mixed-radix bases, the latter step is nontrivial and is better not mixed with other reasoning.

Fig. 5 shows the code and proofs for conversion from associational to positional format, including the fixing of misaligned weights. The helper function `place` takes a pair $t = (w, x)$ (compile-time weight and runtime value) and recursively searches for the highest-index output of the weight function that $w$ is a multiple of. After finding it, `place` returns an index $i$ and the runtime value to be added at that index: $(w/weight_i \cdot x)$. Then, `from_associational` simply adds each limb at the correct index.

The modular-reduction trick we explained in Section III-B can also be succinctly represented in associational format (Fig. 6). First we define `split`, which partitions the list, separating out pairs with compile-time weights that are multiples of the number at which we are splitting (`s`, in the code). Then it divides all the weights that are multiples of `s` by `s` and returns both lists. The `reduce` function simply multiplies the "high" list by `c` and adds the result to the "low" list. Using the `mul` we defined earlier recreates the folklore modular-reduction pattern for prime shapes in which `c` is itself multi-limb.

```
Fixpoint place (t:Z*Z) (i:nat) : nat*Z :=
 if dec (fst t mod weight i = 0) then
  let c:=fst t / weight i in (i, c*snd t)
 else match i with
      | S i' => place t i'
      | O => (O, fst t*snd t) end.

Definition from_associational n p :=
  List.fold_right (λt,
      let wx := place t (pred n) in
      add_to_nth (fst wx) (snd wx) )
    (zeros n) p.

Lemma place_in_range (t:Z*Z) (n:nat)
: (fst (place t n) < S n)%nat.
Proof.
 cbv [place]; induction n; break_match;
 autorewrite with cancel_pair; omega. Qed.

Lemma weight_place t i
: weight (fst(place t i)) * snd(place t i)
  = fst t * snd t.
Proof. (* ... 4 lines elided ... *) Qed.

Lemma eval_from_associational {n} p
: eval (from_associational (S n) p)
  = Associational.eval p.
Proof.
 cbv [from_associational]; induction p;
 push;
 try pose proof place_in_range a n;
 try omega; try nsatz.              Qed.
```

Fig. 5: Coq definition and selected correctness proofs of conversion from associational to positional representation

## E. Carrying

In unsaturated representations, it is not necessary to carry immediately after every addition. For example, with 51-bit limbs on a 64-bit architecture, it would take 13 additions to risk overflow. Choosing which limbs to carry and when is part of the design and is critical for keeping the limb values bounded. Generic operations are easily parameterized on carry strategies, for example "after each multiplication carry from limb 4 to limb 5, then from limb 0 to limb 1, then from limb 5 to limb 6," etc. The library includes a conservative default.

## F. Saturated Arithmetic

The code described so far was designed with unsaturated representations in mind. However, unsaturated-arithmetic performance degrades rapidly when used with moduli that it is not a good match for, so older curves such as P-256 need to be implemented using saturated arithmetic. Basic arithmetic on saturated digits is complicated by the fact that addition and multiplication outputs may not fit in single words, necessitating the use of two-output addition and multiplication

```
Definition split (s:Z) (p:list (Z*Z))
  : list(Z*Z)*list(Z*Z) :=
 let hl :=
   partition (λt, fst t mod s =? 0) p in
 (snd hl,
  map (λt, (fst t / s, snd t)) (fst hl)).
```

```
Definition reduce(s:Z)(c:list _)(p:list _)
  : list(Z*Z) :=
 let lh := split s p in
 fst lh ++ mul c (snd lh).
```

```
Lemma eval_split s p (s_nz:s<>0) :
    eval (fst (split s p))
     + s * eval (snd (split s p))
  = eval p.
Proof. (* ... (7 lines elided) *) Qed.
```

```
Lemma eval_reduce s c p
       (s_nz:s<>0)(m_nz:s-eval c<>0) :
    eval (reduce s c p) mod (s - eval c)
  = eval p mod (s - eval c).
Proof. cbv [reduce]; push.
 rewrite<-reduction_rule,eval_split; auto.
Qed.
```

Fig. 6: Coq code for prime-shape-sensitive modular reduction

```
Definition from_associational
    n (p:list (list Z)) :=
 List.fold_right (λt,
  let p := place t (pred n) in
  cons_to_nth (fst p) (snd p)) (nils n) p.
```

Fig. 7: Conversion from associational to columns

instructions. While the partial products in Fig. 4 also need to be stored in pairs of registers, the halves are never separated, avoiding the need to reason about their representation. On the other hand, saturated-arithmetic algorithms can be seen as treating carry bits (and the low and high halves of a multiplication) as separate limbs. This conceptual framework allows us to reuse most of our unsaturated-associational-arithmetic procedures on saturated representations while still generating the expected low-level code that differs significantly from that of unsaturated arithmetic.

First, we write another version of `mul` that looks very similar to the unsaturated one but uses a two-output multiplication instruction to compute the partial product of $(a, x)$ and $(b, y)$ as `let xy := mul x y in [(a*b, fst xy); (a*b*bound, snd xy)]` instead of `[(a*b, x*y)]`.

Second, rather than using the unsaturated `from_associational`, we first convert to an intermediate representation we call *columns*, named after the conventional layout of partial products during pencil-and-paper multiplication. While associational representations are lists of pairs of integers (`list (Z * Z)`) and positional ones are lists of integers (`list Z`), columns representations are lists of lists of integers (`list (list Z)`). Columns representations are very similar to positional ones, except that the values at each position have not yet been added together. Starting from the columns representation, we can add terms together in a specific order so that carry bits are used right after they are produced, enabling them to be stored in flag registers in assembly code. The code to convert from associational to columns is very similar to associational-to-positional conversion (Fig. 7).

```
Lemma S2_mod
: eval (sat_add S1 (scmul q m)) mod m
  = eval S1 mod m.
Proof. push; zsimplify; auto. Qed.
```

Fig. 8: Proof that adjusting $S$ to be $0 \bmod m$ preserves the correct value $\bmod m$

With these changes in place, much of the code shown before for associational works on positional representations as well. For instance, the partial modular-reduction implementation still applies unchanged; using the new `from_associational` is sufficient. 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*, shown in full earlier in Fig. 1. For moduli $m$ without convenient special structure, there is not much room to optimize the computation of $ab \bmod m$. Instead, it pays off to keep all intermediate values $a$ in a scaled form $aR \bmod m$ for some parameter $R$ and instead optimize the computation of $(aR)(bR)R^{-1} \bmod m = (ab)R \bmod m$. We will use underlines to denote the (unique) Montgomery forms of numbers: $\underline{a} = aR \bmod m$.

First, consider the simple case where $S = \underline{a}\underline{b}$ is divisible by $R$: we can simply divide it by $R$, ignoring the modulus $m$ completely. To make division by $R$ fast, we use a minimal saturated representation for $S$ and pick $R$ to match the weight of some limb. This way, dividing by $R$ needs no code at all: simply forget about the less significant limbs!

Of course, $S$ is extremely unlikely to be divisible by $R$. However, since we only care about the answer $\bmod m$, we can add a multiple of $m$ to adjust $S$ to be $0 \bmod R$. In particular, we want to add $qm$ such that $qm \bmod R = -(S \bmod R)$. Multiplying both sides by $m^{-1} \bmod R$ gives $q = (-m^{-1} \bmod R)(S \bmod R) \bmod R$, which is very cheap to compute: the first factor is a compile-time constant, and $S \bmod R$ is simply the limbs of $S$ below $R$. The Coq proof of this step of our word-by-word implementation is in Fig. 8.

This procedure is further optimized based on the observation that it only acts on the higher limbs of $S$ additively. For $n$-limb $\underline{a}$, instead of first computing $\underline{a}\underline{b}$ and then dividing it by $R = r^n$, it is fine to divide eagerly by $r$ after multiplying $\underline{b}$ with a single limb of $\underline{a}$ – adding the partial products with the

Given $\texttt{eval}\ S < \texttt{eval}\ m + \texttt{eval}\ B$;
To show:

$$\texttt{eval}\ (\texttt{div\_r}\ (\texttt{add}\ (\texttt{add}\ S\ (\texttt{scmul}\ a\ B))\ (\texttt{scmul}\ q\ m))) < \texttt{eval}\ m + \texttt{eval}\ B$$
$$(\texttt{eval}\ S + a * \texttt{eval}\ B + q * \texttt{eval}\ m)/r < \texttt{eval}\ m + \texttt{eval}\ B$$
$$(\texttt{eval}\ m * r + r * \texttt{eval}\ B - 1)/r < \texttt{eval}\ m + \texttt{eval}\ B$$
$$\texttt{eval}\ m + \texttt{eval}\ B - 1 < \texttt{eval}\ m + \texttt{eval}\ B$$

Fig. 9: Intermediate goals of the proof that the word-by-word Montgomery reduction state value remains bounded by a constant, generated by single-stepping our proof script

more significant limbs of $\underline{a}$ commutes with adding multiples of $m$. Even better, adding a multiple of $m$ and a multiple of $\underline{b}$ can only increase the length of $S$ by one limb, balancing out the loss of one limb when dividing $S$ by $r$ and allowing for an implementation that loops through limbs of $\underline{a}$ while operating in-place on $S$. The proof of the last fact is delicate – for example, it is not true that one of the additions always produces a newly nonzero value in the most significant limb and the other never does. In our library, this subtlety is handled by a 15-line/100-word proof script; the high-level steps, shown in Fig. 9, admittedly fail to capture the amount of thought that was required to find the invariant that makes the bounds line up exactly right.

## IV. Certified Bounds Inference

Recall from Section II-B how we use partial evaluation to specialize the functions from the last section to particular parameters. The resulting code is elementary enough that it becomes more practical to apply relatively well-understood ideas from *certified compilers*. That is, as sketched in Section II-C, 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.

### A. Abstract Syntax Trees

The results of partial evaluation fit, with minor massaging, into this intermediate language:

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

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.

Another 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 variable references and type agreement between operators and their arguments.

Several of the key compiler passes 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: $e_1 \ll e_2, e_1 \gg e_2, e_1 \mathbin{\&} e_2, e_1 \mid e_2$
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 only the last three categories, since the earlier ones are familiar from C programming. To chain together multiword 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.

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

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. We do hope, in follow-on work, to prove that a particular compiler preserves the constant-time property in generated assembly. For now, we presume that popular compilers like GCC and Clang do preserve constant time, modulo the usual cycle of occasional bugs and bug fixes. (One additional source-level restriction is important, forcing conditional expressions to be those supported by native processor instructions like conditional move.)

### B. 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.

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

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.

### C. Important Design Choices

Most phases of the compiler use a term encoding called parametric higher-order abstract syntax (PHOAS) [11]. Briefly, this 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 a linear number of simple constraints in propositional logic.

### D. Extensibility with Nonobvious Algebraic Identities

Classic abstract interpretation with intervals works surprisingly well for most of the cryptographic code we have generated. However, a few spans of code call for some algebra to establish the tightest bounds. We considered extending our abstract interpreter with general algebraic simplification, perhaps based on E-graphs as in SMT solvers [12], but in the end we settled on an approach that is both simpler and more accommodating of unusual reasoning patterns. The best-known among our motivating examples is Karatsuba's multiplication, which expresses a multi-digit multiplication in a way with fewer single-word multiplications (relatively slow) but more additions (relatively fast), where bounds checking should recognize algebraic equivalence to a simple multiplication. Naive analysis would treat $(a + b)(c + d) - ac$ equivalently to $(a + b)(c + d) - xy$, where $x$ and $y$ are known to be within the same ranges as $a$ and $c$ but are not necessarily equal to them. The latter expression can underflow. We add a new primitive operator $\mathsf{equivalently}$ such that, within the high-level functional program, we can write the above expression as $\mathsf{equivalently}\ ((a + b)(c + d) - ac, ad + bc + bd)$. The latter argument is used for bounds analysis. The soundness theorem of bounds analysis requires, as a side condition, equality for all pairs of arguments to $\mathsf{equivalently}$. These side conditions are solved automatically during compilation using Coq tactics, in this case `ring` algebra. Compilation then replaces any $\mathsf{equivalently}\ (e_1, e_2)$ with just $e_1$.

## V. Experimental Results

The first purpose of this section is to confirm that implementing optimized algorithms in high-level code and then separately specializing to concrete parameters actually achieves the expected performance. Given the previous sections, this conclusion should not be surprising: as code generation is extremely predictable, it is fair to think of the high-level implementations as simple templates for low-level code. Thus, it would be possible write new high-level code to mimic every low-level implementation. Such a strategy would, of course, be incredibly unsatisfying, although perhaps still worthwhile as a means to reduce effort required for correctness proofs. Thus, we wish additionally to demonstrate that the two simple generic-arithmetic implementations discussed in Sections III-B and III-F are sufficient to achieve good performance in all cases. Furthermore, we do our best to call out differences that enable more specialized implementations to be faster, speculating on how we might modify our framework in the future based on those differences.

## *A. X25519 Scalar Multiplication*

We measure the number of CPU cycles different implementations take to multiply a secret scalar and a public Curve25519 point (represented by the $x$ coordinate in Montgomery coordinates). Despite the end-to-end task posed for this benchmark, we believe the differences between implementations we compare against lie in the field-arithmetic implementation – all use the same scalar-multiplication algorithm and two almost-identical variants of the Montgomery-curve $x$-coordinate differential-addition formulas.

The benchmarks were run on an Intel Broadwell i7-5600U processor in a kernel module with interrupts, power management, Hyper Threading, and Turbo Boost features disabled. Each measurement represents the average of a batch of 15,000 consecutive trials, with time measured using the `RDTSC` instruction and converted to CPU cycles by multiplying by the ratio of CPU and timestamp-counter clock speeds. C code was compiled using `gcc` 7.3 with `-O3 -march=native -mtune=native -fwrapv` (we also tried `clang` 5.0, but it produced ~10% slower code for the implementations here).

| Implementation | CPU cycles | |
|---|---|---|
| `amd64-64`, asm | 151586 | ▬ |
| *this work B*, 64-bit | 152195 | ▬ |
| `sandy2x`, asm | 154313 | ▬ |
| `hacl-star`, 64-bit | 154982 | ▬ |
| `donna64`, 64-bit C | 168502 | ▬ |
| *this work A*, 64-bit | 174637 | ▬ |
| *this work, 32-bit* | 310585 | ▬▬ |
| `donna32`, 32-bit C | 529812 | ▬▬▬▬ |

In order, we compare against `amd64-64` and `sandy2x`, the fastest assembly implementations from SUPERCOP [13] that use scalar and vector instructions respectively; the verified X25519 implementation from the HACL* project [4]; and the best-known high-performance C implementation `curve25519-donna`, in both 64-bit and 32-bit variants. The field arithmetic in both `amd64-64` and `hacl-star` has been verified using SMT solvers [1], [3]. We do not claim that the ranking in this table represents inherent differences between the 5 fastest implementations – we would be entirely unsurprised if a seemingly irrelevant change in the CPU, compiler, or source code rearranged them.

We report on our code generated using the standard representations for both 32-bit and 64-bit, though we are primarily interested in the latter, since we benchmark on a 64-bit processor. We actually report on two of our 64-bit variants, both of which have correctness proofs of the same strength. Variant *A* is the output of the toolchain described in Section III. Variant *B* contains an additional implementation grouping multiplications by the modular-reduction coefficient $c = 19$ with one of the inputs of the multiplication instead of the output $(19 \times (a \times b) \longrightarrow (19 \times a) \times b)$. This algebra pays off because $19 \times a$ can be computed using a 64-bit multiplication, whereas $19 \times (ab)$ would require a significantly more expensive 128-bit multiplication. We currently implement this trick for just $2^{255} - 19$ as an ad-hoc code replacement after partial evaluation but before word-size inference, using the Coq `ring` algebra tactic to prove that the code change does not change the output. We are planning on incorporating a more general version of this optimization into the main pipeline, after a planned refactoring to add automatic translation to continuation-passing style (see Section VII).

The results of a similar benchmark on an earlier version of our pipeline were good enough to convince the maintainers of the BoringSSL library to adopt our methodology, resulting in this Curve25519 code being shipped in Chrome 64 and used by default for TLS connection establishment in other Google products and services. Previously, BoringSSL included the `amd64-64` assembly code and a 32-bit C implementation as a fallback, which was the first to be replaced with our generated code. Then, the idea was raised of taking advantage of lookup tables to optimize certain point ECC multiplications. While the BoringSSL developers had not previously found it worthwhile to modify 64-bit assembly code and review the changes, they made use of our code-generation pipeline (without even consulting us, the tool authors) and installed a new 64-bit C version. The new code (our generated code linked with manually written code using lookup tables) was more than twice as fast as the old version and was easily chosen for adoption, enabling the retirement of `amd64-64` from BoringSSL.

## *B. P-256 Mixed Addition*

Next, we benchmark our Montgomery modular arithmetic as used for in-place point addition on the P-256 elliptic curve with one precomputed input (Jacobian += affine). A scalar-multiplication algorithm using precomputed tables would use some number of these additions depending on the table size. The assembly-language implementation `nistz256` was reported on by Gueron and Krasnov [14] and included in OpenSSL; we also measure its newer counterpart that makes use of the ADX instruction-set extension. The 64-bit C code we benchmark is also from OpenSSL and uses unsaturated-style modular reduction, carefully adding a couple of multiples of the prime each time before performing a reduction step with a negative coefficient to avoid underflow. These P-256 implementations here are unverified. The measurement methodology is the same as for our X25519 benchmarks, except that we did not manage to get `nistz256` running in a kernel module and report userspace measurements instead.

| Implementation | fastest | `clang` | `gcc` | `icc` |
|---|---|---|---|---|
| `nistz256` +ADX | ˜550 | | | |
| `nistz256` AMD64 | ˜650 | | | |
| *this work A* | 1143 | 1811 | 1828 | 1143 |
| OpenSSL, 64-bit C | 1151 | 1151 | 2079 | 1404 |
| *this work B* | 1343 | 1343 | 2784 | 1521 |

Saturated arithmetic is a known weak point of current compilers, resulting in implementors either opting for alternative arithmetic strategies or switching to assembly language. Our programs are not immune to these issues: when we first ran our P-256 code, it produced incorrect output because `gcc` 7.1.1 had generated incorrect code[2]; `clang` 4.0 exited with a mere segmentation fault.[3] Even in later compiler versions where these issues have stopped appearing, the code generated for saturated arithmetic varies a lot between compilers and is obviously suboptimal: for example, there are ample redundant moves of carry flags, perhaps because the compilers do not consider flag registers during register allocation. Furthermore, expressing the same computation using intrinsics such as `_mulx_u64` (variant A in the table) or using `uint128` and a bit shift (variant B) can produce a large performance difference, in different directions on different compilers.

The BoringSSL team had a positive enough experience with adopting our framework for Curve25519 that they decided to use our generated code to replace their P-256 implementation as well. First, they replaced their handwritten 64-bit C implementation. Second, while they had never bothered to write a specialized 32-bit P-256 implementation before, they also generated one with our framework. `nistz256` remains as an option for use in server contexts where performance is critical and where patches can be applied quickly when new bugs are found. The latter is not a purely theoretical concern – Appendix A contains an incomplete list of issues discovered in previous `nistz256` versions.

The two curves thus generated with our framework for BoringSSL together account for over 99% of ECDH connections. Chrome version 65 is the first release to use our P-256 code.

### C. Benchmarking All the Primes

Even though the primes supported by the modular-reduction algorithms we implemented vastly outnumber those actually used for cryptography, we could not resist the temptation to do a breadth-oriented benchmark for plausible cryptographic parameters. The only other implementation of constant-time modular arithmetic for arbitrary modulus sizes we found is the `mpn_sec` section of the GNU Multiple Precision Arithmetic Library[4]. We also include the non-constant-time `mpn` interface for comparison. The list of primes to test was generated by scraping the archives of the ECC discussion list `curves@moderncrypto.org`, finding all supported textual representations of prime numbers.

[2]https://gcc.gnu.org/bugzilla/show_bug.cgi?id=81300, https://gcc.gnu.org/bugzilla/show_bug.cgi?id=81294

[3]https://bugs.llvm.org/show_bug.cgi?id=24943

[4]https://gmplib.org/

One measurement in Fig. 10 corresponds to 1000 sequential computations of a 256-bit Montgomery ladder, aiming to simulate cryptographic use accurately while still constraining the source of performance variation to field arithmetic. A single source file was used for all `mpn_sec`-based implementations; the modulus was specified as a preprocessor macro to allow compiler optimizations to try their best to specialize GMP code. Our arithmetic functions were instantiated using standard heuristics for picking the compile-time parameters based on the shorthand textual representation of the prime, picking a champion configuration out of all cases where correctness proof succeeds, with two unsaturated and one Montgomery-based candidates. The 64-bit trials were run on an x86 Intel Haswell processor and 32-bit trials were run on an ARMv7-A Qualcomm Krait (LG Nexus 4).

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. Appendix B includes the full details, with Tables I and II recording all experimental data points, including for an additional comparison implementation in C++. In our current experiments, compilation of saturated arithmetic in Coq times out for a handful of larger primes. There is no inherent reason for this slowness, but our work-in-progress reimplementation of this pipeline (Section VII) is designed to be executable using the Haskell extraction feature of Coq, aiming to sidestep such performance unpredictability conclusively.

## VI. Related Work

### A. Verified Elliptic-Curve Cryptography

Several past projects have produced verified ECC implementations. We first summarize them and then give a unified comparison against our new work.

Chen et al. [1] verified an assembly implementation of Curve25519, using a mix of automatic SAT solving and manual Coq proof for remaining goals. A later project by Bernstein and Schwabe [2], described as an "alpha test," explores an alternative workflow using the Sage computer-algebra system. This line of work benefits from the chance for direct analysis of common C and assembly programs for unsaturated arithmetic, so that obviously established standards of performance can be maintained.

Vale [5] 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* [4] 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.

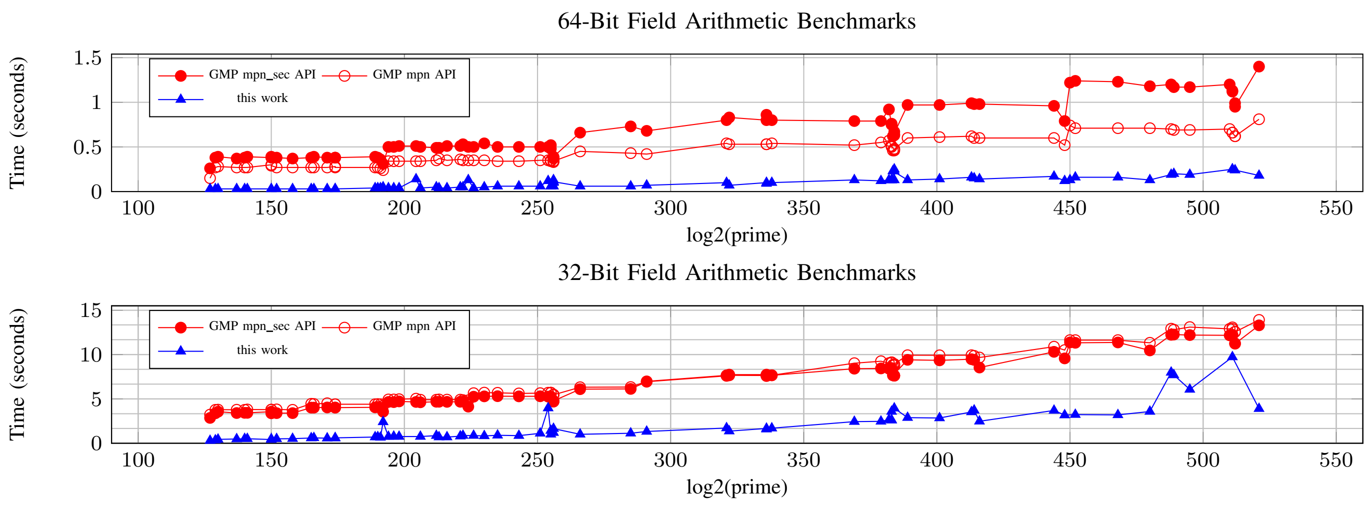

Fig. 10: Performance comparison of our generated C code vs. handwritten using `libgmp`

Primitives are implemented in the Low* imperative subset of F* [15], which supports automatic semantics-preserving translation to C. As a result, while taking advantage of F*'s high-level features for specification, HACL* performs comparably to leading C libraries. Additionally, abstract types are used for secret data to avoid side-channel leaks.

Jasmin [6] 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.

We compare against these past projects in a few main claims.

*Advantage of our work:* **first verified high-performance implementation of P-256**. Though this curve is the most widely used in TLS today, no prior projects had demonstrated performance-competitive versions with correctness proofs. A predecessor system to HACL* [3] verified more curves, including P-256, but in high-level F* code, incurring performance overhead above 100X.

*Advantage of our work:* **code and proof reuse across algorithms and parameters**. A more fundamental appeal of our approach is that it finally enables the high-performance-cryptography domain to benefit from pleasant software-engineering practices of abstraction and code reuse. The code becomes easier to understand and maintain (and therefore to prove), and we also gain the ability to compile fast versions of new algorithm variants automatically, with correctness proofs. Considering the cited past work, it is perhaps worth emphasizing how big of a shift it is to do implementation and proving at a parameterized level: as far as we are aware, no past project on high-performance verified ECC has ever *verified implementations of multiple elliptic curves* or *verified multiple implementations of one curve, targeted at different hardware characteristics*. With our framework, all of those variants can be generated automatically, at low developer cost; and indeed the BoringSSL team took advantage of that flexibility.

*Advantage of our work:* **small trusted code base**. Every past project we mentioned includes either an SMT solver or a computer-algebra system in the trusted code base. Furthermore, each past project also trusts some program-verification tool that processes program syntax and outputs symbolic proof obligations. We trust only the standard Coq theorem prover, which has a proof-checking kernel significantly smaller than the dependencies just mentioned, and the (rather short, whiteboard-level) statements of the formal claims that we make (plus, for now, the C compiler; see below).

*Disadvantage of our work:* **stopping at C rather than assembly**. Several past projects apply to assembly code rather than C code. As a result, they can achieve higher performance and remove the C compiler from the trusted base. Our generated code is already rather close to assembly, differing only in the need for some register allocation and instruction scheduling. In fact, it seems worth pointing out that every phase of our code-generation pipeline is necessary to get code *low-level enough* to be accepted as input by Jasmin or Vale. We have to admit to nontrivial frustration with popular C compilers in the crypto domain, keeping us from using them to bridge this gap with low performance cost. We wish compiler authors would follow the maxim that, if a programmer takes handwritten assembly and translates it line-by-line to the compiler's input language, the compiler should not generate code slower than the original assembly. Unfortunately, no popular compiler seems to obey that maxim today, so we are looking into writing our own (with correctness proof) or integrating with one of the projects mentioned above.

*Disadvantage of our work:* **constant-time guarantee mechanized only for straightline code**. Our development also includes higher-level cryptographic code that calls the primitives discribed here, with little innovation beyond comprehensive functional-correctness proofs. That code is written following "constant-time" coding rules and thus should be free of timing side channels barring human error. However, we do not yet do certified compilation or static analysis for timing leaks on the higher-level code. To this end, it could be advantageous to take inspiration from, or adopt, one of the tools mentioned above.

### *B. Other Related Work*

Performance-oriented synthesis of domain-specific code (without proofs of correctness) has previously been done using explicit templates (e.g. Template Haskell [16]) and more sophisticated multistage programming (e.g. Lightweight Modular Staging (LMS) [17]). More specialized frameworks along these lines include FFTW [18] and Spiral [19]. Out of these, our strategy is most similar to the one from LMS, differing mainly in the choice of using existing (proof-generating) Coq facilities for partial evaluation and rewriting rather than implementing them ourselves. While verified compilers (e.g., CakeML [20], CompCert [21]) and translation validators [22] are useful for creating soundly optimized versions of a reference program, we are not aware of any that could cope with abstraction-collapsing synthesis as done in this work or LMS.

Myreen and Curello verified a general-purpose big-integer library [23]. The code uses a single uniform base system, does not include specialized modular-reduction optimizations, and does not run in constant time. However, their verification extends down to AMD64 assembly using verified decompilation.

Verification of cryptographic protocols (e.g., FCF [24], CertiCrypt [25]) 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 elliptic-curve primitives used in TLS) and that the protocol is secure (out of scope for this paper). The work by Beringer et al. [26] 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.

## VII. Discussion

We would like to remark on the aspects of elliptic-curve-cryptography implementation that made this approach work as well as it did, to aid future application in other contexts. The most general (and perhaps the most important) takeaway is that effort put into structuring code in the most instructive manner possible pays off double during verification, enough to justify the development of new tooling to make that code run fast. In cases where generalizing an algorithm makes its operation and invariants more apparent, we think it simply makes sense to prove correctness for the general version and use partial evaluation to derive the desired code, even if a specialized implementation has already been written.

On the other hand, we do not believe that the phase distinction between arithmetic and bounds analysis is fundamental, although it is convenient. When viewing uniform positional representations as polynomials, this distinction corresponds to finding a bound on the degree of the polynomial using partial evaluation and finding bounds on its coefficients using certified compilation, but cryptographic implementation literature (e.g. [10]) has treated degree and coefficient ranges rather similarly. We are aware of one example that would be simpler if the high-level algorithms operated on tuples $(l, x, h)$ s.t. $l \leq x \leq h$ instead of arbitrary-precision integers $x$. The BoringSSL P-256 implementation that was replaced with our code used prime-shape-specific modular reduction with negative $c_i$ and avoided limb underflow by adding the modulus to a field element before any addition where the input limb ranges did not already rule out underflow. A certified compiler that needs to preserve the exact value of each limb cannot use this strategy, but it could be encoded in the framework so far by figuring out when to add the modulus for balance in some other way in the high-level algorithm.

The bounds-analysis pipeline described in this paper only works on straight-line code, greatly simplifying the implementation. It is geared towards being used on the bodies of inner loops where naively picking the largest available integer type for each variable would be costly, but temporaries are numerous enough that picking sizes by hand would be tedious and error-prone. The analysis could be generalized to loops and recursion by iterating until convergence, but it is unclear whether that is called for in any application.

While we do appreciate being able to support a number of plausibly useful elliptic curves with the same (parametrized) code and proof, we believe that engineering benefits, especially simplicity of code and proof, make this approach worth considering even when targeting a single configuration.

We find it arbitrary and unfortunate that arithmetic algorithms need to be written in continuation-passing style. As CPS was only necessary for using Coq's term reduction for partial evaluation, writing a certified partial evaluator would be sufficient to avoid it. Preliminary experiments have given promising results, but we have not finalized the interface yet. We also expect that replacing the controlled term reduction with a certified compilation pass would significantly improve the compilation time.

We would like to shrink our trusted base by connecting to a certified compiler targeting assembly. However, existing compilers are not smart enough at mapping the data flow between instructions onto time and space available on the hardware. Thus another fruitful future-work area is studying those optimizations, principally combined register allocation and instruction scheduling, even independently of proof.

## Acknowledgments

This work was supported in part by a Google Research Award and National Science Foundation grants CCF-1253229, CCF-1512611, and CCF-1521584. We benefited greatly from a fruitful collaboration with Google involving David Benjamin, Thai Duong, Adam Langley, Dominic Rizzo, and Marius Schilder. Robert Sloan contributed to this project as a student at MIT, before joining Google. We thank Jason Donenfeld for teaching us how to benchmark arithmetic code with Linux kernel modules, as well as for setting up benchmarks for popular Curve25519 implementations. For comments on drafts of the paper, we thank Tej Chajed, Istvan Chung, Karl Samuel

Gruetter, Ivan Kuraj, Adam Langley, Derek Leung, Devin Neal, Rahul Sridhar, Peng Wang, Ray Wang, and Daniel Ziegler.

## References

[1] 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*. ACM, 2014, pp. 299–309, document ID: 55ab8668ce87d857c02a5b2d56d7da38. [Online]. Available: http://cryptojedi.org/papers/#verify25519

[2] D. J. Bernstein and P. Schwabe, 2016. [Online]. Available: http://gfverif.cryptojedi.org/

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

[4] J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche, "HACL*: A verified modern cryptographic library," in *Proc. CCS*, 2017. [Online]. Available: https://eprint.iacr.org/2017/536.pdf

[5] B. Bond, C. Hawblitzel, M. Kapritsos, R. Leino, J. Lorch, B. Parno, A. Rane, S. Setty, and L. Thompson, "Vale: Verifying high-performance cryptographic assembly code," in *Proc. USENIX Security*, 2017. [Online]. Available: http://www.cs.cornell.edu/~laurejt/papers/vale-2017.pdf

[6] J. B. Almeida, M. Barbosa, G. Barthe, A. Blot, B. Grégoire, V. Laporte, T. Oliveira, H. Pacheco, B. Schmidt, and P.-Y. Strub, "Jasmin: High-assurance and high-speed cryptography," in *Proc. CCS*, 2017.

[7] "Web browsers by version (global marketshare)." [Online]. Available: https://clicky.com/marketshare/global/web-browsers/versions

[8] D. Benjamin, in personal communication about TLS connections initiated by Chrome, 2017.

[9] S. Gueron, "Efficient software implementations of modular exponentiation," 2011. [Online]. Available: https://eprint.iacr.org/2011/239.pdf

[10] 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, 2006. [Online]. Available: http://cr.yp.to/papers.html#curve25519

[11] 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. [Online]. Available: http://adam.chlipala.net/papers/PhoasICFP08/

[12] D. Detlefs, G. Nelson, and J. B. Saxe, "Simplify: a theorem prover for program checking," *Journal of the ACM*, vol. 52, no. 3, pp. 365–473, 2005.

[13] D. J. Bernstein and T. Lange, "eBACS: ECRYPT benchmarking of cryptographic systems," 2017. [Online]. Available: https://bench.cr.yp.to/supercop/supercop-20170228.tar.xz

[14] S. Gueron and V. Krasnov, "Fast prime field elliptic curve cryptography with 256 bit primes," 2013. [Online]. Available: https://eprint.iacr.org/2013/816.pdf

[15] J. Protzenko, J.-K. Zinzindohoué, A. Rastogi, T. Ramananandro, P. Wang, S. Zanella-Béguelin, A. Delignat-Lavaud, C. Hriţcu, K. Bhargavan, C. Fournet, and N. Swamy, "Verified low-level programming embedded in F*," *Proc. ACM Program. Lang.*, vol. 1, no. ICFP, pp. 17:1–17:29, Aug. 2017. [Online]. Available: http://doi.acm.org/10.1145/3110261

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

[17] T. Rompf and M. Odersky, "Lightweight modular staging: A pragmatic approach to runtime code generation and compiled DSLs," *Proceedings of GPCE*, 2010. [Online]. Available: https://infoscience.epfl.ch/record/150347/files/gpce63-rompf.pdf

[18] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005, special issue on "Program Generation, Optimization, and Platform Adaptation". [Online]. Available: http://www.fftw.org/fftw-paper-ieee.pdf

[19] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo, "SPIRAL: Code generation for DSP transforms," *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, vol. 93, no. 2, pp. 232–275, 2005. [Online]. Available: https://users.ece.cmu.edu/~moura/papers/ieeeproceedings-pueschelmouraetal-feb05-ieeexplore.pdf

[20] 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*. ACM Press, Jan. 2014, pp. 179–191. [Online]. Available: https://cakeml.org/popl14.pdf

[21] X. Leroy, "A formally verified compiler back-end," *J. Autom. Reason.*, vol. 43, no. 4, pp. 363–446, Dec. 2009. [Online]. Available: http://gallium.inria.fr/~xleroy/publi/compcert-backend.pdf

[22] G. C. Necula, "Translation validation for an optimizing compiler," in *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, ser. PLDI '00. New York, NY, USA: ACM, 2000, pp. 83–94. [Online]. Available: https://people.eecs.berkeley.edu/~necula/Papers/tv_pldi00.pdf

[23] M. O. Myreen and G. Curello, "A verified bignum implementation in x86-64 machine code," in *Proc. CPP*, 2013. [Online]. Available: http://www.cse.chalmers.se/~myreen/cpp13.pdf

[24] A. Petcher and G. Morrisett, "The Foundational Cryptography Framework," in *Proceedings of the 4th International Conference on Principles of Security and Trust - Volume 9036*. New York, NY, USA: Springer-Verlag New York, Inc., 2015, pp. 53–72. [Online]. Available: http://adam.petcher.net/papers/FCF.pdf

[25] G. Barthe, B. Grégoire, and S. Zanella-Béguelin, "Formal certification of code-based cryptographic proofs," in *36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009*. ACM, 2009, pp. 90–101. [Online]. Available: http://software.imdea.org/~szanella/Zanella.2009.POPL.pdf

[26] L. Beringer, A. Petcher, K. Q. Ye, and A. W. Appel, "Verified correctness and security of OpenSSL HMAC," in *24th USENIX Security Symposium*, Aug. 2015, pp. 207–221. [Online]. Available: https://www.usenix.org/system/files/conference/usenixsecurity15/sec15-paper-beringer.pdf

[27] D. J. Bernstein, B. van Gastel, W. Janssen, T. Lange, P. Schwabe, and S. Smetsers, "TweetNaCl: A crypto library in 100 tweets," in *Progress in Cryptology – LATINCRYPT 2014*, ser. Lecture Notes in Computer Science, D. Aranha and A. Menezes, Eds., vol. 8895. Springer-Verlag Berlin Heidelberg, 2015, pp. 64–83, document ID: c74b5bbf605ba02ad8d9e49f04aca9a2. [Online]. Available: http://cryptojedi.org/papers/#tweetnacl

[28] B. Brumley, M. B. M. Barbosa, D. Page, and F. R. G. Vercauteren, "Practical realisation and elimination of an ECC-related software bug attack," 2011. [Online]. Available: https://eprint.iacr.org/2011/633.pdf

## Appendix A
## Study of Bugs in Similar Crypto Code

Highly optimized handwritten cryptographic arithmetic implementations have an uninspiring history of serious bugs, even when written and audited by respected experts. To get a sense of the details, we surveyed project bug trackers and other Internet sources, stopping after finding 26 bugs (each hyperlinked to its bug report) in implementation of nontrivial cryptography-specific optimizations. Fig. 11 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 is a part of 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 these 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 refers 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` [27, 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 $10 \cdot 2^{-29}$ [28].
- Mathematical misconceptions: The CryptoNote double-spending bug arose from use of an algorithm on a composite-order elliptic curve when it is only applicable in a prime-order group.

## Appendix B
## Full Results of Many-Primes Experiments

Tables I and II 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 `curves@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 multiple 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 the C compiler 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.

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

| Reference/*Category* | 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 | $x^2 \bmod m$ | Montgomery form, AMD64 assembly | carry, exploitable |
| openssl#1593 | P384 modular reduction | | carry, exploitable |
| tweetnacl-U32 | irrelevant | bit-twiddly C | assumed 32-bit `long` |
| *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 | P-256 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 |
| 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? |
| bitcoin#eed71d85 | ECDSA-secp256k1 x*B | mixed addition Jacobian+Affine | missing case |
| *Elliptic Curves* | | | |
| ed25519.py | Ed25519 | accepts signatures other impls reject | missing $h \bmod l$ |
| openjdk#01781d7e | EC scalarmult | mixed addition Jacobian+Affine | missing case |
| jose-adobe | ECDH-ES | 5 different libraries affected | 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 | $\infty$ 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$(P) \neq l$ |

Fig. 11: Survey of bugs in algebra-based cryptography implementations

| **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 I: 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 | | Speed -up |
|---|---|---|---|---|---|
| | Sol. | 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 II: 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 four times as many operations. The C++ GMP program was not benchmarked on 32-bit.