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

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

Andres Erbsen
*MIT CSAIL*
andreser@mit.edu

Jade Philipoom
*MIT CSAIL*
jadep@mit.edu

Jason Gross
*MIT CSAIL*
jgross@mit.edu

Robert Sloan
*MIT CSAIL*
rob.sloan@alum.mit.edu

Adam Chlipala
*MIT CSAIL*
adamc@csail.mit.edu

## Abstract

We introduce an unusual 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 such initial code into highly competitive 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 a wide range of 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.

This is an abridged version of the full paper originally presented in IEEE S&P 2019 [10]. We have omitted most proof-engineering details in favor of a focus on the system's functional capabilities.

## 1 Introduction

As verification makes the transition from research topic to component of real-world software ecosystems, the field must face challenges of scale, in terms of both amount of code and conceptual complexity. Each system, of course, has its own constraints and objectives, and these needs will best be met with a broad toolbox of approaches. In this paper, we sketch a case study following a relatively unusual approach, in the hope that it will help illuminate the broad array of possibilities for verification projects. The full paper was originally presented in IEEE S&P 2019 [10]; what follows is a version abridged, selectively updated, and adapted for a more systems-oriented audience.

That audience is probably most familiar with projects fitting the classic mold of proving theorems about programs written by hand, as exemplified by sEL4 [16], IronClad [13, 14], FSCQ [5, 6, 15], CertiKOS [11], and various case studies in push-button verification [17, 18, 20, 21]. Often the implementers of these projects do rewrite systems from scratch in tandem with proofs, but the final code artifacts implement very similar pseudocode to off-the-shelf systems. A moderately widely known alternative is *program synthesis*, which writes a program automatically from its logical specification. Most colloquially, this idea is associated with relatively intensive combinatorial search through spaces of possibilities, often taking advantage of SMT solvers like Z3 [8]. However, some systems-code domains are focused enough that we can write very predictable and *verified* generic implementations. With the help of a verified domain-specific compiler, we can translate these general problem descriptions into practical, performance-competitive code. This paper considers a successful example of the template-based code-generation strategy.

Our project verifies efficient modular-arithmetic routines for elliptic-curve cryptography, a domain to which much optimization effort has been applied. This means that competitive implementations use highly specialized techniques, previously accessible only to a small number of experts. We aimed to create a system that generalized these approaches and proved, with minimal trusted code, that the detailed low-level routines implemented their high-level arithmetic specifications. To that end, we first studied handwritten implementations and extracted fully general descriptions of the complex design choices they required. Then, we encoded these algorithms in the Coq proof assistant [22] as templates, generic over compile-time parameters like the modulus and integer width of the underlying architecture. Once these parameters are plugged in, a domain-specific compiler transforms the templates into efficient low-level code.

The output code is encoded in a specialized low-level abstract syntax tree (AST) that includes only very basic fixed-width integer operations, at the approximate abstraction level

of assembly instructions, and is easily translatable to a number of output languages. At current count we can produce C, Go, and Rust, along with two other unusual specialized languages for specific applications. The C code, our primary target, is competitive with existing handwritten C, although it still falls short of the performance of handwritten assembly. However, our toolchain, with no input other than the compile-time parameters, produces efficient and completely verified code, and we believe it presents a very interesting case study in a verification approach that is unusual in *generating* code, rather than working backwards from existing code. (It is worthwhile to note here that one can still verify specific existing code by proving it equivalent to the generated code, and the equivalence is much more straightforward than equivalence with a high-level specification, and we have used this technique to verify existing production code.) Our verification stretches all the way from very low-level C to the natural high-level specifications of modular arithmetic (e.g. $\forall x, y.\ \text{add}(x, y) = (x + y) \bmod p$). The output code uses a restricted set of instructions by construction, preventing timing side channels. The overall trusted computing base includes the Coq proof checker, a simple pretty-printer, and the C language toolchain.

In addition to being interesting research, our approach has proven to be practically applicable, as it 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 implementations of Curve25519 and P-256, the two most widely used elliptic curves via Internet standards. 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 has about 60% market share [1], and 90% of connections use X25519 or P-256 [2]). 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.

We believe our approach is particularly suited to design domains that a) involve methodical low-level transformations that are nonetheless finicky and easy for humans to get wrong, and b) apply these transformations to a wide range of setups with different parameters. Elliptic-curve field arithmetic fits this description, because most efficient implementations use the same family of optimization techniques, while there are hundreds of combinations of moduli and native-integer sizes. A change to either one usually dramatically changes the resulting code, even though the underlying techniques are similar. Other opportunities to apply these ideas may include families of very closely related data structures being used in OS kernels and other systems software.

This abridged paper will be a whirlwind tour through the most interesting and informative aspects of our approach. First, Section 2 will go over a common optimization technique for modular reduction and integer representation, as an example of the kinds of algorithms our templates encode. Then, in Section 3, we will explain the stages of our domain-specific compiler and its capabilities. Finally, Section 4 will show a sample of our benchmarks. For more details, particularly for proof-engineering techniques, explorations of related work, and full benchmarks, please look to the full-length paper [10]. For even *more* details, the entire development is called **Fiat Cryptography**, and is available under the MIT license at:

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

## 2 Arithmetic Template Library

In this section, we will describe the arithmetic optimization strategies we encoded in our template library. 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 an implementation of modular reduction specialized to take advantage of the specific shape of its fixed modulus. 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 correctness certificates.

### 2.1 Multi-Limbed Arithmetic

Cryptographic modular arithmetic implementations distribute very large numbers across smaller *limbs* of 32- or 64-bit integers. Fig. 1 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*. This abridged paper will only describe the unsaturated arithmetic in our development; for saturated arithmetic, please refer to the full paper.

Another interesting feature shown in the examples is that the exponents of some bases, such as the original 32-bit Curve25519 representation [3], 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.

Why accept all of this added complication, and waste available bits, instead of just using saturated representations for everything? These unorthodox big integers are fast primarily because of a specific modular-reduction algorithm, pseudo-Mersenne reduction, which becomes extremely fast when the number of bits in the prime *corresponds to a limb boundary*.

| prime | architecture | # limbs | base | representation (distributing large $x$ into $x_0 \ldots x_n$) |
|---|---|---|---|---|
| $2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$ (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 | $2^{25.5}$ | $x = x_0 + 2^{26}x_1 + 2^{51}x_2 + 2^{77}x_3 + \ldots + 2^{204}x_8 + 2^{230}x_9$ |
| $2^{448} - 2^{224} - 1$ (p448) | 64-bit | 8 | $2^{56}$ | $x = x_0 + 2^{56}x_1 + 2^{112}x_2 + \ldots + 2^{392}x_7$ |
| $2^{127} - 1$ | 64-bit | 3 | $2^{42.5}$ | $x = x_0 + 2^{43}x_1 + 2^{85}x_2$ |

Figure 1: Examples of big-integer representations for different primes and integer widths

$$\begin{aligned} s \times t = 1 \times s_0t_0 + 2^{43} \times s_0t_1 \quad & + 2^{85} \times s_0t_2 \\ + 2^{43} \times s_1t_0 \quad & + 2^{86} \times s_1t_1 \quad + 2^{128} \times s_1t_2 \\ & + 2^{85} \times s_2t_0 \quad + 2^{128} \times s_2t_1 \quad + 2^{170} \times s_2t_2 \\ = s_0t_0 \quad & + 2^{43}(s_0t_1 + s_1t_0) + 2^{85}(s_0t_2 + 2s_1t_1 + s_2t_0) + 2^{127}(2s_1t_2 + 2s_2t_1) + 2^{170} \times s_2t_2 \end{aligned}$$

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

For instance, reduction modulo $2^{255} - 19$ is fastest when the 255th bit of a bignum is the first bit of a limb. In a saturated implementation, limb boundaries would fall on multiples of the integer size (for instance, a 64-bit saturated representation would have limb boundaries at bits 0, 64, 128, etc.). But in an unsaturated implementation, with a funky base like $2^{25.5}$, we get limb boundaries at bits 0, 26, 51, 77, ... and 255. The vast speed improvement in modular reduction is worth keeping a few extra registers around to represent each large number.

## 2.2 A Note on 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.

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

To give a more concrete sense of how the representation described above works in practice, we will walk through a modular multiplication procedure specialized to the (relatively) small modulus $2^{127} - 1$ and a 64-bit architecture. Say we want to multiply two numbers $s$ and $t$ in its field, with those inputs broken up into limbs 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. 2.

We have formatted the calculation 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 the multiplication step is complete, but we still need to do modular reduction; we don't want our answer to have five limbs when it should need only three. Here we use the pseudo-Mersenne reduction trick: $2^k \bmod (2^k - c) = c$, so $a + 2^k b \equiv a + cb \pmod{2^k - c}$. We "divide" the last two limbs of the product, the terms with weights $2^{127}$ and $2^{170}$, by $2^{127}$. Because we chose our base system wisely, this division is actually a no-op; we just now consider these limbs to have different weights. This is the big trick that makes pseudo-Mersenne reduction fast: eliminating division by clever choice of base system. Then we can multiply the high limbs by $c$, which in this case is 1, and add them to the first three digits, providing the following final formula:

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

Note that this is a *partial* modular-reduction algorithm; it is not guaranteed that the result is less than $2^{127} - 1$, just that the result fits in three limbs.

In the code, `src/Demo.v` includes another example that walks through modular multiplication with pseudo-Mersenne reduction, using the modulus from Curve25519 [3]. The full generic template library is found under `src/Arithmetic`.

## 3 Domain-Specific Compiler

### 3.1 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 our library of high-level functional implementations that generalize the patterns lurking behind hand-written implementations 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 $\mathsf{add}\ [a_1,a_2,a_3]\ [b_1,b_2,b_3]$, 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 could use Coq's term-reduction machinery, which allows us to choose λ-calculus-style reduction rules to apply until reaching a normal form. Here is what happens with our example, when we ask Coq to leave let and + unreduced but apply other rules. (Note that :: is a notation for appending to the front of a list.)

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

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 work around this issue, we chose to implement this phase of our pipeline as a certified compiler[1]. 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.

Our certified compiler handles partial evaluation and let-lifting, turning this function into the straightline code

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

[1]The compiler was ongoing work at the time of our original publication and was only briefly discussed as a future improvement; in the original paper, we described using Coq's built-in term-reduction machinery. However, although the proof-engineering techniques have changed dramatically, the function is the same.

Chaining together sequences of function calls leads to idiomatic and efficient straightline code, 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.

### 3.2 Word-Size Inference

Up to this point, we have derived code that looks almost exactly like the C 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.

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

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

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 too as another phase in our certified compiler.

### 3.3 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. But this is the entire extent of the compromise: as described in the next section, we have been able to match or exceed the performance of all C code we sought to replicate.

## 4 Experimental Results

The 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. We will detail how this works out for the two most prominent applications of our framework, the TLS ECC fields modulo $2^{255}-19$ and $2^{256}-2^{224}+2^{192}+2^{96}-1$. In our full paper, we additionally demonstrate that the two simple templates generalized from these case studies are sufficient to achieve good performance across a broad sample of finite fields proposed for elliptic-curve-cryptography use.

### 4.1 X25519 Scalar Multiplication

We measured 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.

The benchmarks were run using `gcc` 7.3 on an Intel Broadwell i7-5600U processor in a kernel module with interrupts, power management, Hyper Threading, and Turbo Boost features disabled. We are presenting them as a rather arbitrary illustration of the performance that can be achieved using our approach; it is expected that the relative speeds could differ by as much as a couple of times across different microarchitectures, compilers, and compiler flags.

| Implementation | CPU cycles | |
|---|---|---|
| `OLHFR`, asm | 121444 | |
| `amd64-64`, asm | 151586 | |
| *this work*, 64-bit | 152195 | |
| `sandy2x`, asm | 154313 | |
| `hacl-star`, 64-bit | 154982 | |
| `donna64`, 64-bit C | 168502 | |

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. In order, we compare against `OLHFR`, the non-precomputation-based implementation from [19]; `amd64-64` and `sandy2x`, the fastest assembly implementations from SUPERCOP [4] that use scalar and vector instructions respectively; the verified X25519 implementation from the HACL* project [24]; 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 [7,23]. Our code is generated from a general template for unsaturated pseudo-mersenne arithmetic which is proven correct for all parameters. We previously applied a few optimizations to $2^{255}-19$ specifically, but have since integrated the generalized forms of these into our template.

The results of a similar benchmark on an early prototype of our methodology were good enough to convince the maintainers of the BoringSSL library to adopt it, resulting in this Curve25519 code being shipped since 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.

### 4.2 P-256 Mixed Addition

Next, we benchmarked 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 [12] and included in OpenSSL; we also measured 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. The same pattern is also present for X25519, although less pronounced: the two fastest assembly implementations listed earlier use a 4-limb saturated representation, but the speedup over 5-limb unsaturated assembly is smaller than the slowdown incurred in C code due to heavy use of carry flags. 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 – the appendices of our full paper contain a sample 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 initiated by Google Chrome.

## 5 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.

Looking towards the future, we would like to extend our pipeline to bypass the C compiler and target assembly, shrinking our trusted code base. This would require studying combined register allocation and instruction scheduling; since our output AST is very low-level, the C compiler is doing little else for us. Alternatively, hand-optimized assembly code could be checked to implement the same machine-word-level computation as our generated code using an equality-checking engine like those found in modern SMT solvers (e.g., [9]), allowing verification of optimizations that are out-of-reach for compilers, for example effective use of vector instructions.

[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

## 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 Daniel J. Bernstein, 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] Web browsers by version (global marketshare). https://clicky.com/marketshare/global/web-browsers/versions.

[2] David Benjamin. in personal communication about TLS connections initiated by Chrome, 2017.

[3] Daniel 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.

[4] Daniel J. Bernstein and Tanja Lange. eBACS: ECRYPT benchmarking of cryptographic systems. 2017.

[5] Haogang Chen, Tej Chajed, Alex Konradi, Stephanie Wang, Atalay Ileri, Adam Chlipala, Frans Kaashoek, and Nickolai Zeldovich. Verifying a high-performance crash-safe file system using a tree specification. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, October 2017.

[6] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, Frans Kaashoek, and Nickolai Zeldovich. Using Crash Hoare logic for certifying the FSCQ file system. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, October 2015.

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

[8] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *TACAS*, 2008.

[9] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *Journal of the ACM*, 52(3):365–473, 2005.

[10] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. Simple high-level code for cryptographic arithmetic – with proofs, without compromises. In *IEEE Security & Privacy*, May 2019.

[11] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. CertiKOS: an extensible architecture for building certified concurrent OS kernels. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX - Advanced Computing Systems Association, October 2016.

[12] Shay Gueron and Vlad Krasnov. Fast prime field elliptic curve cryptography with 256 bit primes, 2013.

[13] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jay Lorch, Bryan Parno, Lilith Stephenson, Srinath Setty, and Brian Zill. IronFleet: Proving practical distributed systems correct. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*. ACM - Association for Computing Machinery, October 2015.

[14] Chris Hawblitzel, Jon Howell, Jay Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. Ironclad apps: End-to-end security via automated full-system verification. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX - Advanced Computing Systems Association, October 2014.

[15] Atalay Ileri, Tej Chajed, Adam Chlipala, Frans Kaashoek, and Nickolai Zeldovich. Proving confidentiality in a file system using DiskSec. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, October 2018.

[16] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 207–220. ACM, 2009.

[17] Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. Scaling symbolic evaluation for automated verification of systems code with Serval. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*. ACM - Association for Computing Machinery, October 2019.

[18] Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. Hyperkernel: Push-button verification of an OS kernel. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*. ACM - Association for Computing Machinery, October 2017.

[19] Thomaz Oliveira, Julio López, Hüseyin Hışıl, Armando Faz-Hernández, and Francisco Rodríguez-Henríquez. How to (pre-)compute a ladder. In Carlisle Adams and Jan Camenisch, editors, *Selected Areas in Cryptography – SAC 2017: 24th International Conference, Ottawa, Ontario, Canada, August 16 - 18, 2017, Revised Selected Papers*, pages 172–191. Springer International Publishing, 2018.

[20] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. Push-button verification of file systems via crash refinement. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX - Advanced Computing Systems Association, October 2016.

[21] Helgi Sigurbjarnarson, Luke Nelson, Bruno Castro-Karney, James Bornholt, Emina Torlak, and Xi Wang. Nickel: A framework for design and verification of information flow control systems. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX - Advanced Computing Systems Association, October 2018.

[22] The Coq Development Team. The Coq proof assistant, version 8.10.0, October 2019.

[23] Jean Karim Zinzindohoue, Evmorfia-Iro Bartzia, and Karthikeyan Bhargavan. A verified extensible library of elliptic curves. In *IEEE Computer Security Foundations Symposium (CSF)*, 2016.

[24] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. HACL*: A verified modern cryptographic library. In *Proc. CCS*, 2017.