High-assurance field inversion for curve-based cryptography

Benjamin S. Hvass, Diego F. Aranha and Bas Spitters

Concordium Blockchain Research Center (COBRA)
Department of Computer Science, Aarhus University, Denmark
{bsh,dfaranha,spitters}@cs.au.dk

Abstract. Modern cryptography must satisfy a myriad of security properties, ranging from sound hardness assumptions to correct and secure implementations that resist side-channel cryptanalysis. Curve-based cryptography is not different in this regard, and substantial progress in the last few decades has been achieved in both selecting parameters and devising secure implementation strategies. In this context, the security of implementations of field inversion is sometimes overlooked in the research literature, because (i) the approach based on Fermat's Little Theorem (FLT) suffices performance-wise for many parameters used in practice; (ii) it is typically invoked only at the very end of scalar multiplication or pairing computation, with a small impact in performance; (iii) it is challenging to implement securely for general parameters without a significant performance penalty. However, field inversion can process sensitive information and must be protected with side-channel countermeasures like any other cryptographic operation, as illustrated by recent attacks [ASS17, AGTB18, AGB20]. In this work, we focus on *timing attacks* against field inversion for primes of cryptographic interest, both in the case when FLT-based inversion can be efficiently implemented or not. We extend the Fiat-Cryptography framework, which synthesizes provably correct-by-construction implementations, to implement the Bernstein-Yang constant-time inversion algorithm as a step toward this goal. This allows a correct implementation of prime field inversion to be conveniently synthesized for any prime. We benchmark the implementations across a range of primes for curve-based cryptography and they outperform traditional FLT-based approaches in most cases, with observed speedups up to 2.5 for the largest parameters. Our work is already used in production in the MirageOS unikernel operating system[1].

Keywords: Field inversion · Constant-time execution · Implementation security · Formal verification.

1 Introduction

Finite field arithmetic is pervasive in number-theoretic public-key cryptography, and an essential ingredient of Elliptic Curve Cryptography (ECC) and Pairing-based Cryptography (PBC). In many cases, its implementation dictates how efficient and secure the overall cryptosystem behaves in practice. The field inversion operation is a peculiar case, since it is rarely among the performance-critical portions of the implementation and most efficient algorithms for the general case are hard to implement securely without a massive performance penalty [Bos14]. For this reason, field inversion is often implemented using the Fermat's Little Theorem (FLT) approach of exponentiating by $p-2$ in $\mathbb{F}_p$ for prime p. This is an efficient strategy for ECC implementations relying on special primes with fast modular reduction, especially when the exponent allows a short addition chain [Ber06]. When

[1] https://hannes.robur.coop/Posts/EC

performance is a more pressing concern or parameters are not friendly to FLT inversion, implementers typically resort to using a variable-time version of the Extended Euclidean Algorithm (EEA). However, bugs and side-channel leakage in the EEA implementation can lead to attacks against RSA [AGTB18] and ECC [ASS17, AGB20]. These are not just threats of research interest, for example a vulnerability recently discovered on the EEA implementation in Windows could be exploited to mount denial of service attacks [2].

Field arithmetic in Montgomery representation, as commonly found in PBC, is a particularly challenging case for field inversion. The FLT approach is not favored by the *dense* prime moduli in popular pairing-friendly families of curves [BN05, BD17], in combination with slower modular reduction through Montgomery's arithmetic [Mon85]. Although field inversion is not performance-critical for the pairing computation itself, it arises during exponentiation in pairing groups and unlocks the compressed squaring optimization for the final exponentiation [AKL+11]. With pairings being increasingly deployed as a fundamental building block for zero-knowledge proofs and privacy-preserving cryptocurrencies [BLS04, BCG+14], the threat of implementation bugs becomes more important, as they can allow attacks which may compromise the security and privacy guarantees of these cryptographic systems ([BBPV12, lfS17, EPG+19]).

In order to satisfy performance constraints, current efficient software implementations of ECC and PBC rely on hand-optimized architecture-specific Assembly code for the underlying field arithmetic and a great deal of manual tuning to unlock the best performance across a range of architectures [AFK+12, ABLR13]. This introduces low-level code which is both hard to audit and to verify as correct. Moreover, implementations need at least to be *constant-time*, in the sense that execution time does not depend on input and protection against timing attacks is provided given some performance penalty. As an illustrating case using the popular BLS12-381 curve for motivation, the cost of one scalar multiplication in the base curve is reported to be around 382,000 cycles on Intel Skylake [Ara17] using variable-time inversion. According to our benchmarking in Table 3, a constant-time field inversion using publicly available code would add at least 200,000 cycles to that figure for the two required conversions to affine coordinates (one for the table of precomputed points, the other for the result). This impact is prohibitive and motivates the need for more efficient alternatives.

Recent progress in the literature allows this problem to be solved elegantly. Bernstein and Yang proposed in 2019 a constant-time Euclidean algorithm based on *division steps* that can be generalized for polynomial arithmetic, comes with a mathematical proof and is surprisingly efficient for field inversion [BY19]. In that same year, an alternative path for implementing cryptographic libraries was demonstrated as viable in the Fiat-Crypto framework [EPG+19]. By combining correct-by-construction optimized low-level code with automatically generated and formally verified high-level code, it became possible to develop libraries which are both efficient and formally verified. Unfortunately, Fiat-Crypto does not provide an inversion operation and the implementer must build its own approach based on the other field operations, creating the same risk of insufficient *post-hoc* analysis.

Our contributions. We extend the Fiat-Crypto framework with a constant-time implementation of field inversion based on the Bernstein-Yang approach of iterating division steps. We implement the original version of the algorithm (with the jumpdivstep optimization) and the "half-deta" variant, recently developed to optimize ECDSA signing over the curve `secp256k1` [WMea21] adopted in Bitcoin. This variant requires a lower number of division steps to be evaluated, which immediately translates to performance improvement.

Our work completes the set of finite field operations which Fiat-Crypto supports, and consists in the first efficient verified implementation of field inversion for several primes, including those needed for PBC. Moreover, it allows to conveniently synthesize a correct and

[2] https://bugs.chromium.org/p/project-zero/issues/detail?id=1804

portable implementation of the algorithm for *any* prime using the two main representations supported in Fiat-Crypto (unsaturated Solinas and Montgomery). We observe that the Montgomery representation is less competitive with hand-written code when compiled with GCC, however. Our formulation of the algorithm maximally relies on what is provided by Fiat-Crypto. In particular, we take advantage of the field operations provided by the framework whenever possible, instead of introducing custom new operations. In the context of Montgomery arithmetic, this introduces expensive modular multiplications to update the matrix coefficients, the effects of which we mitigate by employing the lazy reduction optimization and adjusting the precomputed constant. According to our benchmarks, we achieve a performance penalty of up to 4.3 compared to our own unverified constant-time Assembly-accelerated implementations of inversion for a range of parameters in both ECC and PBC settings from 254 to 575 bits. This slowdown can be tolerable if correctness is of critical importance or if inversion performance is less critical. For the PBC primes, our implementation consistently outperforms the FLT approach accelerated with finite field arithmetic in Assembly, with speedups ranging from 1.8 to 2.5 for different sizes. For the ECC primes, we outperform the FLT approaches in the two largest parameters and achieve performance improvement of up to 25% against an implementation based on Fiat-Crypto and 5% against the hand-written Assembly counterpart.

In comparison with the FLT approach for inversion, our implementation provides a performance improvement for all PBC fields employing Mongtomery representation, even when the FLT implementations are accelerated by record-setting hand-written Assembly. Our work is already used in production in the MirageOS unikernel[3], showing that it is fast enough for industry projects with a focus on correctness.

Outline of the paper We will briefly explain the necessary preliminaries of Fiat-Crypto and the inversion algorithm in Sections 2 and 3. Sections 4 and 5 describe our implementation of the algorithm in Fiat and our formalization of the correctness proof, respectively.

2 Fiat Cryptography

Fiat Cryptography[EPG+19] (or just Fiat-Crypto) is a framework for generating verified finite field arithmetic which is correct by design. The approach was illustrated through the implementation of field arithmetic for several standardized elliptic curves using an extensible code generation framework, capable of producing code competitive in performance with popular hand-optimized multi-precision libraries. It provides a simple CLI which takes a prime and a machine word size and generates C source files implementing most finite field operations necessary to implement e.g. elliptic curve cryptography. Java, Go and Rust are also supported. Code generated by Fiat-Crypto is currently being used in production in Firefox[4], BoringSSL[5] and the WireGuard VPN[6].

In Fiat-Crypto there are separate binaries to generate code for each style of multi-precision arithmetic: Montgomery, saturated and unsaturated representations. Fiat-Crypto does not formally prove that the generated code is constant time, but only generates "straight-line code", i.e. code without branching that should run in constant-time after it is processed by an optimizing compiler. The verification steps are conducted using the Coq proof assistant [Tea20]. It consists of a verified compiler from a subset of Coq to a

[3]https://github.com/mirage/mirage-crypto/tree/main/ec/native
[4]https://blog.mozilla.org/security/2020/07/06/performance-improvements-via-formally-verified-cryptography-in-firefox/
[5]https://boringssl.googlesource.com/boringssl/+/master/third_party/fiat/
[6]https://www.wireguard.com/formal-verification/

simple language embedded in Coq containing only bitwise and machine-integer operations. From here, the generated terms can be pretty-printed to several programming languages.

Correct-by-construction vs. verifying existing implementations. Fiat-Crypto differs from other verification projects in a significant way: Instead of verifying an existing implementation against a specification, it provides a pipeline for generating verified implementations. This has the advantage of only requiring a single formalization effort. Verification of complex software is a laborious procedure, so in many cases it will not be deemed important enough. Having auto-generated code allows verified code to be used in such cases. Another advantage is the support for multiple languages; in general each implementation in a different language would present a separate formalization effort.

Coq. Coq [Tea20] is a state-of-the-art interactive proof assistant based on dependent type theory. A proof assistant is software that allows one to construct proofs, and in particular, proofs about (functional) programs. Coq reduces all proofs to a small *kernel* — it is thus *foundational* in that it reduces everything to the axioms of mathematics. Next to its build-in functional programming language, Coq also has a more ad-hoc scripting language for *tactics*. Users write tactics to direct Coq to construct, or search for, proofs. When, after a list of such tactic instructions, the proof is fully completed it is finally checked by the kernel for correctness.

We will use the Coq standard library throughout this article. In particular, we use the standard implementations of (unary) naturals `nat` and (binary and infinite precision) integers `Z`.

Multi-precision arithmetic. For cryptographic purposes, it usually necessary to do arithmetic on numbers much larger than a single machine word. These are usually represented using arrays of digits and interpreted as a number in some large radix size (e.g. a full word size). We will refer to the entries of these arrays as *limbs* and numbers represented as such as *multi-limb numbers*.

In Fiat-Crypto multi-limb integers are represented as lists of integers, i.e. as the type `list Z`. Such a list of numbers, say `[1;12;123]`, corresponds to sum of its elements up to some weighing of the indices, e.g. $1 \cdot 2^{\text{weight } 0} + 12 \cdot 2^{\text{weight } 1} + 123 \cdot 2^{\text{weight } 2}$, where `weight` is some map from `nat` to `int`. Note that the representation is low-endian. When reasoning about multi-limb numbers, one uses the function `eval` to evaluate the number as an integer by adding together its limbs (multiplied by their respective weights).

We will refer to representations using a full-word radix as *saturated*. When doing arithmetic on such representation one has to take care of propagating carries, as additions do not fit within one register. Conversely, we will refer to a radix smaller than a full word size as *unsaturated*. We will refer to arithmetic on these numbers as *multi-precision arithmetic*, as opposed to *single-precision arithmetic*, which we will assume is implemented natively in the platform.

There are a variety of optimizations and algorithms for multi-precision arithmetic, and more precisely for multi-precision *modular* arithmetic modulo some large number, as used in cryptography. One of the more expensive operations in modular arithmetic is reduction, as it generally requires a multi-precision division. Reduction is necessary after a multi-precision multiplication or squaring. We will briefly describe two specialized ways of reducing which are both used in our implementations.

For integers a, b and c we write $a \equiv b \pmod{c}$ when c divides the difference between a and b. For integers a, c we write $a \bmod c$ for the unique integer b between 0 and c satisfying $a \equiv b \pmod{c}$.

Generalized Mersenne Reduction If the modulus M is of the form $2^k + c_1 2^{k-1} + \cdots + c_k$ for some integers k and c_i (which satisfies some constraints [Sol99]), then M is said to be a *generalized Mersenne number* (or *Solinas number*). In that case there is an improved algorithm for reduction which replaces division with a linear number of additions and shifting operations.

Depending on the coefficients and exponents of the integral polynomial this can be more or less efficient, but we will not go into details here. A notable example of a generalized Mersenne number which is used in cryptographic implementations is the prime $2^{255} - 19$ over which the elliptic curve `curve25519` is defined.

Montgomery Reduction Next to Generalized Mersenne Reduction, Fiat-Crypto also supports Montgomery arithmetic [Mon85]. If R is a number coprime to the modulus M, then the *Montgomery reduction* modulo M of a number a is the number $aR^{-1} \bmod M$. Montgomery reduction can be computed more efficiently than generic reduction when R is chosen appropriately. The algorithm performs divisions by R instead of M, so R is chosen such that divisions are cheap. For example, by choosing R to be a power of 2 such divisions become simple shifts.

The factor R^{-1} might look out of place, but Montgomery reduction can be used when computing multiplications by working in the "Montgomery domain", which simply means operations are performed on numbers multiplied by R. That is, to compute $ab \bmod M$ we instead compute $(aR \bmod M)(bR \bmod M)$ and compute a Montgomery reduction. We obtain $(aR \bmod M)(bR \bmod M)R^{-1} \bmod M = abR \bmod M$, the product in the Montgomery domain. This achieves modular multiplication without divisions.

Multiplying with $R \bmod M$ every time might seem expensive, but if multiple arithmetic operations can be performed before converting back again, then this cost becomes negligible. One can also add naturally in the Montgomery domain, since

$$(aR \bmod M + bR \bmod M) \bmod M = (a+b)R \bmod M.$$

Because Montgomery reduction has the same complexity of a multi-precision multiplication, another popular optimization in Montgomery arithmetic is *lazy reduction*, which adds unreduced multiplication results (up to $M \times R$) before a full reduction is needed.

3 Bernstein-Yang inversion

The Bernstein-Yang (BY) inversion algorithm [BY19] is a new and competitive constant-time algorithm for inverting in finite fields. A special case of this is for the fields $\mathbb{Z}/p\mathbb{Z}$. The algorithm is a constant time variant of the classical Extended Euclidean Algorithm (EEA). We implement (Section 4) and formalize (Section 5) the BY algorithm in Fiat-Crypto.

3.1 Specification and correctness

The algorithm uses a division step (divstep), which we define for all integers δ, g and odd integers f as

$$\operatorname{divstep}(\delta, f, g) = \begin{cases} \left(1-\delta, g, \frac{g-f}{2}\right) & \text{if } \delta > 0 \text{ and } g \text{ odd} \\ \left(1+\delta, f, \frac{g+(g \bmod 2)f}{2}\right) & \text{otherwise.} \end{cases} \tag{1}$$

The requirement that f is odd makes divstep an endofunction on $\mathbb{Z} \times \mathbb{Z} \times (2\mathbb{Z}+1)$. The branch can be implemented in constant time and thus so can divstep.

Algorithm 1: divsteps

```
Input   : Integers $n, \delta, f$ and $g$ such that $f$ is odd
Output  : The integers $\delta_n, f_n$ and $g_n$ and the matrix product $\mathcal{T}_n\mathcal{T}_{n-1}\cdots\mathcal{T}_0$
1  $u \leftarrow 1$, $v \leftarrow 0$, $q \leftarrow 0$, $r \leftarrow 1$ ;
2  for $i \leftarrow 1$ to $n$ do
3      if $0 < \delta$ and $g$ odd then
4          $\delta \leftarrow -\delta$, $f \leftarrow g$, $g \leftarrow -f$, $u \leftarrow q$, $v \leftarrow r$, $q \leftarrow -u$, $r \leftarrow -v$ ;
5      $g_0 \leftarrow g \bmod 2$ ;
6      $\delta \leftarrow \delta + 1$ ;
7      $g \leftarrow \frac{g+g_0 f}{2}$ ;
8      $u \leftarrow 2u$ ;
9      $v \leftarrow 2v$ ;
10     $q \leftarrow q + g_0 u$ ;
11     $r \leftarrow r + g_0 v$ ;
12 return $\delta, f, g, \begin{pmatrix} u & v \\ q & r \end{pmatrix}$
```

Algorithm 2: BY-inversion

```
Input   : Integers $f$ and $g$ such that $f$ is odd and $\gcd(f,g) = 1$
Output  : Integer $g^{-1}$ such that $gg^{-1} = 1 \pmod f$
1  $d \leftarrow \max(\log_2 f, \log_2 g)$ ;
2  if $d < 46$ then
3      $m \leftarrow \lfloor (49d + 80)/17 \rfloor$ ;
4  else
5      $m \leftarrow \lfloor (49d + 57)/17 \rfloor$ ;
6  $e \leftarrow ((f+1)/2)^m \bmod f$ ;
7  $\delta \leftarrow 1$ ;
8  $\delta, f, g, \begin{pmatrix} u & v \\ q & r \end{pmatrix} \leftarrow \mathrm{divsteps}(m, \delta, f, g)$ ;
9  $g^{-1} \leftarrow e \cdot v \cdot \operatorname{sgn} f$ ;
```

We will also use the following *transition matrices*

$$\mathcal{T}(\delta, f, g) = \begin{cases} \begin{pmatrix} 0 & 2 \\ -1 & 1 \end{pmatrix} & \text{if } \delta > 0 \text{ and } g \text{ odd} \\ \begin{pmatrix} 2 & 0 \\ g \bmod 2 & 1 \end{pmatrix} & \text{otherwise.} \end{cases} \tag{2}$$

These are transition matrices in the sense that multiplication captures applying divstep once (up to a factor; see also theorem 9.1 in [BY19]).

To compute the inverse of g modulo f we will need to iterate the divstep, compute the transition matrix of the resulting values and sequentially multiply these matrices. This procedure is captured in alg. 1. For integers δ, f and g we write $(\delta_n, f_n, g_n) = \mathrm{divstep}^n(\delta, f, g)$ and $\mathcal{T}_n = \mathcal{T}(\delta_n, f_n, g_n)$.

The divstep procedure can then be used to implement modular inversion as described in alg. 2. To implement field inversion for a fixed modulus, we can precompute d, m and e in the algorithm. The algorithm does precomputations (lines 1-7), iterates division steps a constant number of times (line 8) and combines the two (line 9); where $\operatorname{sgn}(\cdot)$ computes the sign of an integer.

The correctness of this algorithm is summarized in the following theorem:

Theorem 1 (Theorem 11.2 in [BY19]). *Let f and g be integers with f odd. Let d be a real number such that $f^2 + 4g^2 \leq 5 \cdot 2^{2d}$. Let m be an integer such that $m \geq \lfloor (49d + 80)/17 \rfloor$ if $d < 46$ and $m \geq \lfloor (49d + 57)/17 \rfloor$ if $d \geq 46$.*

For $i = 1, 2, \ldots, m$, let $(\delta_i, f_i, g_i) = \mathrm{divstep}^i(1, f, g)$ and $\mathcal{T}_i = \mathcal{T}(\delta_i, f_i, g_i)$ and $\begin{pmatrix} u_i & v_i \\ q_i & r_i \end{pmatrix} = \mathcal{T}_{i-1}\mathcal{T}_{i-2} \cdots \mathcal{T}_0$. Then $g_m = 0$, $f_m = \pm \gcd(f, g)$ and $v_m g = 2^m f_m \pmod{f}$.

The correctness of alg. 2 follows from Theorem 1 since f and g are assumed to be coprime, the final values of f and v are f_m and v_m, and p is the inverse of 2^m modulo f, so

$$p \cdot v \cdot (\operatorname{sgn} f) \cdot g = (2^m)^{-1} \cdot v \cdot (\pm 1) \cdot g = (\pm 1) \cdot (\pm 1) = 1 \pmod{f}$$

as required.

The theorem as stated here differs slightly from the one in [BY19] since our definition of $\mathcal{T}$ is scaled by a factor to avoid having to reason about rational numbers.

3.2 Outline of proof

The proof of Theorem 1 is in 4 parts:

- Specification of a related algorithm for computing the gcd of two numbers.
- Complexity analysis of the related algorithm.
- Applying the complexity of the gcd algorithm to bound the amount of divsteps needed before reaching a fixpoint.
- Proving that reaching a fixpoint of divstep yields the modular inverse.

Respectively, these are described in appendix E, F, G and section 11 in [BY19].

We will expand on how each part was formalized in Section 5. For the proofs we need the definition of *2-adic valuation.* If g is an integer and p is a prime then the *p-adic valuation* of g is the highest power of p which divides g. We will denote it by $\mathrm{ord}_p\, g$ or $\mathrm{val}_p\, g$ (in the literature ν_p is also common). We will also write $\mathrm{split}_p\, g$ for g divided by this maximal power of p, i.e. $\mathrm{split}_p\, g = g/p^{\mathrm{ord}_p\, g}$.

While the paper proof usually uses 2-adic integers, we only use the corresponding statements for integers. This facilitates the formalization and suffices to prove Theorem 1. Using 2-adic integers does however lead to some more general results, so one could imagine redoing the formalization in this more general setting.

3.3 The jumpdivstep optimization

Algorithm 1 can be optimized by observing that computing the k first iterations of divsteps only depends on the k first bits in f and g. This allows us to work on smaller numbers and "jump" through the divsteps computation in larger steps.

Algorithm 3: The jumpdivstep algoritm

```
Input  : Integers $n, k, \delta, f$ and $g$ such that $f$ is odd and $k \mid n$
Output : The integers $\delta_n, f_n$ and $g_n$ and the matrix product $\mathcal{T}_n\mathcal{T}_{n-1}\cdots\mathcal{T}_0$
1 $\mathcal{T} \leftarrow \begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix}$ ;
2 for $i \leftarrow 1$ to $n/k$ do
3     $f' \leftarrow f \bmod 2^k, g' \leftarrow g \bmod 2^k$ ;
4     $\delta, f', g', \mathcal{T}' \leftarrow \mathrm{divsteps}(k, \delta, f', g')$ ;
5     $\begin{pmatrix}f\\ g\end{pmatrix} \leftarrow \mathcal{T}\begin{pmatrix}f\\ g\end{pmatrix}/2^k$ ;
6     $\mathcal{T} \leftarrow \mathcal{T}' \cdot \mathcal{T}$ ;
7 return $\delta, f, g, \mathcal{T}$
```

One way to see that this is correct is to note that one run through the loop corresponds to k runs through the loop in alg. 1 (where the matrix $\mathcal{T}$ corresponds to the four variables u, v, q and r). Indeed, which branch is chosen in divsteps for the first k iterations only depends on the first k bits of f and g, since the the $k-1$ first bits in $(g-f)/2$ and $(g + (g \bmod 2)f)/2$ (the two possibilities for next g-value) only depend on the k first bits of g and f.

The concrete values of f and g have no influence on u, v, q and r. So, the matrix we get in line 4 of alg. 3 is indeed correct and we multiply with the current product in line 6 as required. Now, to see that the updated values of f and g are correct, simply note that for integers $i \leq j$

$$2^{j-i}\begin{pmatrix}f_j\\ g_j\end{pmatrix} = \mathcal{T}_{j-1}\mathcal{T}_{j-2}\cdots\mathcal{T}_i\begin{pmatrix}f_i\\ g_i\end{pmatrix}$$

which follows from the fact that $2\begin{pmatrix}f_{i+1}\\ g_{i+1}\end{pmatrix} = \mathcal{T}_i\begin{pmatrix}f_i\\ g_i\end{pmatrix}$ (by definition) and induction (see also theorem 9.1 in [BY19]). We already established that $\mathcal{T}'$ is equal to the intermediate matrix product. So, f' and g' are equal to f_{ik} and g_{ik} (in the i'th iteration).

4 Verified and efficient field inversion in Fiat-Crypto

Our implementation of alg. 1 was already integrated to Fiat-Crypto, and alg. 3 will be submitted to the main repository after peer review. Meanwhile, we provide a forked version of the library containing the implementations[7]. The generated code for programming language `lang` can be found in folder `fiat-lang`, and standalone testing/benchmarking programs for illustration can be found in the folder `inversion-c` together with a `Makefile`. All file paths in this section will be relative to this root. To implement the Bernstein-Yang algorithm in Fiat-Crypto we needed to add several primitives to the framework. The implementation is verified by relating it to the algorithm formalized in Section 5.

A major part of specifying and implementing the algorithm was implementing and formalizing *signed* multi-precision arithmetic for the types of f and g in alg. 1, since this was absent from the framework. Our main contributions to the repository are the files `src/Arithmetic/BYInv.v` and `src/Arithmetic/BYInvJump.v`. Beyond these files, we also made several additions to the `src/Util/ZUtil` folder (mainly theorems about two's complement representations of integers, e.g. `TwosComplement.v`, `TwosComplementMul.v`, `SignExtend.v` etc.). Furthermore integrating the implementations from `BYInv.v` and `BYInvJump.v` into the compilation pipeline required some additions (see e.g. the files `src-`

[7]https://github.com/bshvass/fiat-crypto

/PushButtonSynthesis/BYInversionReificationCache.v, src/PushButtonSynthesis-/WordByWordMontgomery.v and src/PushButtonSynthesis/UnsaturatedSolinas.v.

In the following m will refer to the machine word size for which the implementation is parameterized. In Fiat-crypto word-sized integers are represented as infinite precision integers, i.e. Z, and multi-limb integers are represented as lists of word sized integers in Coq, i.e. list Z.

When programming in Fiat-Crypto, one has to use the supported low-level language, i.e. the language whose terms it can compile into the embedded C-like language and consequently generate the code. Notable supported operations are bitwise operations on integers: >> for right shifts, |' for bitwise or and |' for bitwise and. Furthermore, there is Z.lnot_modulo which interprets a number as of some bit-length and then flips all bits in its binary representation.

We will use the functions that are already implemented in the framework whenever we can. Important examples include Z.zselect, a constant time selection implementation on word sized integers, select, a constant time selection implementation on multi-limb integers, and several multi-precision modular arithmetic implementations.

4.1 Representing signed word sized integers

We use the following definition to represent the numbers from -2^m to $2^m - 1$

```
Definition twos_complement m a :=
  (if ((a mod 2 ^ m) <? 2 ^ (m - 1)) then a mod 2 ^ m else a mod 2 ^ m - 2 ^ m).
```

Then e.g. twos_complement m (2 ^ m - 1) = -1 as usual for a two's-complement representation. We need three operations on this representation to be able to implement the algorithms.

Arithmetic right shift To get efficient division by powers of two we implement division as shifts and since we have signed integers the shift has to preserve the most significant bit. We will need shifts for both alg. 1 and alg. 3. We implement it as follows

```
Definition arithmetic_shiftr m a k :=
  dlet q := Z.zselect (sign_bit m a) 0 (ones_from m k) in
        q |' (a >> k).
```

where the sign_bit and ones_from are defined as

```
Definition sign_bit m a := a >> (m - 1).
Definition ones_from m k := (Z.ones k) << (m - k).
```

where Z.ones k simply is $2^k - 1$ (k ones in binary).

We prove it correct, in the sense that arithmetically shifting an integer and interpreting it in two's complement corresponds to first interpreting it and then dividing by an appropriate power of 2.

```
Lemma arithmetic_shiftr_spec m a k
      (Hm : 0 < m)
      (Ha : 0 <= a < 2 ^ m)
      (Hk : 0 <= k) :
  Z.twos_complement m (Z.arithmetic_shiftr m a k) = (Z.twos_complement m a) / 2 ^ k.
```

Negation To compute $-a$ for an integer a in two's complement, you simply flip all bits of the number and add 1. In Coq we get the definition

```
Definition twos_complement_opp m a :=
  ((Z.lnot_modulo a (2 ^ m)) + 1) mod (2 ^ m).
```

and correctness

```
Lemma twos_complement_opp_spec m a
      (Hm : 0 < m)
      (corner_case : Z.twos_complement m a <> - 2 ^ (m - 1)) :
  Z.twos_complement m (Z.twos_complement_opp m a) = - (Z.twos_complement m a).
```

Note that the `corner_case` assumption is quite natural, since -2^{m-1}'s inverse 2^{m-1} cannot be represented with m bits (in two's complement).

Comparison to zero Computing $0 \leq a$ for an integer a in two's complement is done by checking the most significant bit. What we need to compute is $0 < a$, and since $0 < a$ if and only if $-a \leq 0$, we simply check the most significant bit of $-a$.

```
Definition twos_complement_pos m a :=
dlet b := twos_complement_opp m a in sign_bit m b.
```

The corresponding correctness of this operation is

```
Lemma twos_complement_pos_spec m a
      (mw0 : 0 < m)
      (corner_case : Z.twos_complement m a <> - 2 ^ (m - 1)) :
  Z.twos_complement_pos m a = Z.b2z (0 <? Z.twos_complement m a).
```

Here `Z.b2z` is simply the map on booleans sending `false` to 0 and `true` to 1.

4.2 Representing signed multi-limb integers

Since we will need signed multi-limbs integers, we define our own extended evaluation to reason about these, which also interprets a list as a number in two's complement. This is simply first evaluating as a regular multi-limb number and then interpreting it in two's complement, i.e.

```
Definition eval_twos_complement machine_wordsize n f :=
  Z.twos_complement (machine_wordsize * Z.of_nat n) (eval (uweight machine_wordsize) n f).
```

where `uweight` is a saturated weight function and `n` is the amount of limbs needed, i.e. the length of f.

We will need the following operations on signed multi-limb numbers.

Arithmetic right shift As for word sized integers, we will need to divide multi-limb numbers by powers of two.

```
Definition sat_arithmetic_shiftr machine_wordsize n f k :=
  (map
     (fun i =>
        ((nth_default 0 f i) >> k) |' (Z.truncating_shiftl machine_wordsize
                                                          (nth_default 0 f (i + 1))
                                                          (machine_wordsize - k)))
     (seq 0 (n - 1)))
    ++ [Z.arithmetic_shiftr machine_wordsize (nth_default 0 f (n - 1)) k].
```

The function `Z.truncating_shiftl` shifts an integer to left, but truncates at a specified bit width. It is equivalent to left shifting and then reducing mod 2^m, where m is the specified bit width. The function `nth_default` takes a default value, a list and an index as arguments and returns the element of the list at the index unless the index is out of bounds; in that case it returns the default value. The expression `seq n m` constructs the list of consecutive integers of length `m` integers starting at `n`, e.g. `seq 2 3 = [2,3,4]`. Thus, the function picks out each element in f, shifts them to the right and changes their k most significant bits to the k least significant bits of the following element, except for the last element which it simply arithmetically shifts to the right.

The correctness theorem is as follows

```
Lemma eval_sat_arithmetic_shiftr machine_wordsize n f k
      (Hn : (0 < n)%nat)
      (Hf : length f = n)
      (Hk : 0 <= k < machine_wordsize)
      (Hf2 : forall z, In z f -> 0 <= z < 2^machine_wordsize) :
  eval_twos_complement machine_wordsize n
                        (sat_arithmetic_shiftr machine_wordsize n f k) =
  eval_twos_complement machine_wordsize n f / (2 ^ k).
```

For correctness, the shifted amount must be less than a word size. Fiat-Crypto **cannot** compile this implementation. When using this for generating code one has to instantiate k with some constant, i.e., you can only generate shifts by a constant. This suffices to implement the Bernstein-Yang algorithm.

Negation To negate we flip all bits of all limbs in the list and then use our addition with carry to add one. This gives the inverse as in the word sized case.

```
Definition sat_opp machine_wordsize n f :=
  sat_add machine_wordsize n
          (sat_one n)
          (map (fun i => Z.lnot_modulo i (2^machine_wordsize)) f).
```

Here is the correctness of the implementation.

```
Lemma eval_twos_complement_sat_opp machine_wordsize n f
      (mw0 : 0 < machine_wordsize)
      (n0 : (0 < n)%nat)
      (Hz : forall z, In z f -> 0 <= z < 2^machine_wordsize)
      (Hf : length f = n)
      (corner_case : eval_twos_complement machine_wordsize n f <> - 2 ^ (machine_wordsize * n - 1)):
  eval_twos_complement machine_wordsize n (sat_opp machine_wordsize n f) =
  - eval_twos_complement machine_wordsize n f.
```

Addition We also need addition but this is already implemented in the Fiat-Crypto library, so we simply take it from there. Note that we do not need subtraction, since we can do this by adding the inverse (which is computed by `sat_opp`).

Parity checking Computing the parity of a multi-limb integer is straight forward: It is simply checking the parity of the least significant limb.

```
Definition sat_mod2 f := nth_default 0 f 0 mod 2.
```

4.3 Implementing divsteps

The implementation of divsteps is in the file `src/Arithmetic/BYInv.v`. There is both a definition for the Montgomery arithmetic and Unsaturated Solinas. In the same file we prove that the implementation computes a divstep as specified in Section 3.1. We have included the implementation using Montgomery style arithmetic here in Fig. 1.

The implementation uses modular arithmetic for u, v, q and r variables since these would otherwise grow much larger than necessary (by Theorem 1 we only need these numbers modulo f), regular signed multiple-precision arithmetic for f and g and word sized arithmetic for δ.

We have already explained most of the functions used but some remain. Function `addmod` is multiple-precision modular addition and `oppmod` is multiple-precision modular negation. These were already implemented in Fiat-Crypto. The correctness theorem of `divstep` is `divstep_correct_full` in `src/Arithmetic/BYInv.v` which asserts that the method computes a divstep.

Note that the implementation does include the loop in alg. 1. The issue with loops is that the only way to get Fiat-Crypto to generate them is to unroll them. However, the

```
Definition divstep_aux (machine_wordsize : Z)
                       (sat_limbs mont_limbs : nat)
                       (m : Z)
                       (data : Z * (list Z) * (list Z) * (list Z) * (list Z)) :=
  let '(d,f,g,v,r) := data in
  dlet cond := Z.land (twos_complement_pos' machine_wordsize d) (sat_mod2 g) in
  dlet d' := Z.zselect cond d (twos_complement_opp' machine_wordsize d) in
  dlet f' := select cond f g in
  dlet g' := select cond g (sat_opp machine_wordsize sat_limbs f) in
  dlet v' := select cond v r in
  let v'':= addmod machine_wordsize mont_limbs m v' v' in
  dlet r' := select cond r (oppmod machine_wordsize mont_limbs m v) in
  dlet g0 := sat_mod2 g' in
  let d'' := (fst (Z.add_get_carry_full (2^machine_wordsize) d' 1)) in
  dlet f'' := select g0 (sat_zero sat_limbs) f' in
  let g'' := sat_arithmetic_shiftr1 machine_wordsize sat_limbs
                                    (sat_add machine_wordsize sat_limbs g' f'') in
  dlet v''' := select g0 (sat_zero mont_limbs) v' in
  let r'' := addmod machine_wordsize mont_limbs m r' v''' in
  (d'',f',g'',v'',r'').
```

Figure 1: Implementation of a divstep in Fiat-Crypto

body of the loop is already very large so unrolling the entire loop for even moderately large primes becomes unreasonable.

As of now, one has to write the loop manually after generating the code, with small effort and low risk of making mistakes. There are templates for this in `src/inversion-c`. We have included the template for Montgomery arithmetic in Fig. 2. As discussed later, another solution could be to use more advanced tools to generate the loop and reason about its correctness (see Section 7).

4.4 Implementing jumpdivsteps

The implementation of jumpdivsteps (alg. 3) is in the file `src/Arithmetic/BYInvJump.v`. There is both a definition for the Montgomery arithmetic and Unsaturated Solinas. We have included the implementation in Montgomery arithmetic here in Fig. 3.

For jumpdivsteps we need a couple of additional methods. The idea of jumpdivsteps is that we can compute the divsteps (in line 4) on word sized integers (we use $k = m - 2$ such that all intermediate values in divstep fit in a word). This however also means that the entries of the result matrix $\mathcal{T}'$ are word sized integers and thus we have to multiply word sized and multi-limb numbers when computing the matrix vector product in line 5. This functionality was basically already implemented in Fiat-Crypto, but we wrapped it in `word_sat_mul`.

Also, the numbers in $\mathcal{T}$ have to be modular reduced (otherwise they grow too large), so when we have to compute the matrix product in line 6, we have to reduce the entries of $\mathcal{T}$ modulo f (they might for instance be negative). This is what `twos_complement_word_to_montgomery_no_encode` does. It does this by computing the negation and then choosing based on sign (recall that it has to compute the negation always otherwise a branch is introduced). This conversion does **not** convert into the Montgomery domain; we avoid doing this to save several expensive multiplications. We simply accumulate these factors and multiply them after the loop.

The correctness of `outer_loop_body` is `outer_loop_body_correct`, which expresses the fact that computing `outer_loop_body` amounts to computing `m-2` division steps.

As in the case of regular divsteps, one also has to write the loop manually after generating the code for jumpdivsteps, but only for the outer loop though. We do in fact unroll the inner divsteps computation (in line 4 in alg. 3). This is what the `fold_right`

```c
void inverse(WORD out[LIMBS], WORD g[SAT_LIMBS]) {

  WORD precomp[LIMBS];
  PRECOMP(precomp);

  WORD d = 1;
  WORD f[SAT_LIMBS];
  WORD v[LIMBS];
  WORD r[LIMBS];
  WORD out1;
  WORD out2[SAT_LIMBS], out3[SAT_LIMBS], out4[LIMBS], out5[LIMBS];

  MSAT(f);
  MONE(r);
  for (int j = 0; j < LIMBS; j++) v[j] = 0;

  for (int i = 0; i < ITERATIONS - (ITERATIONS % 2); i+=2) {
    DIVSTEP(&out1,out2,out3,out4,out5,d,f,g,v,r);
    DIVSTEP(&d,f,g,v,r,out1,out2,out3,out4,out5);
  }
  if (ITERATIONS % 2) {
    DIVSTEP(&out1,out2,out3,out4,out5,d,f,g,v,r);
    for (int k = 0; k < LIMBS; k++) v[k] = out4[k];
    for (int k = 0; k < SAT_LIMBS; k++) f[k] = out2[k];
  }

  WORD h[LIMBS];
  OPP(h, v);
  SZNZ(v, f[SAT_LIMBS -1 ] >> (WORDSIZE - 1), v, h);
  MUL(out, v, precomp);

  return;
}
```

Figure 2: Handwritten loop to implement alg. 2 from alg. 1; all functions called are generated by Fiat.

term achieves in Fig. 3.

Note that in the source files there are two implementations of jump divstep (`jump_template.c` and `jump_alt_template.c`). The “alt” version is the one we have discussed here, which generates the entire body of the jump divstep loop. The other implementation requires one to write the body by hand and only generates the required functions. Since this requires more handwritten code and the observed performance is the approximately same, we regard the “alt”-version as the optimal one. There are however two caveats: (i) the size of the generated code implementing the entire loop body is quite big since it has to be unrolled for Fiat to generate it (this is worse for larger primes) and (ii) the time needed for code generation grows by a non-trivial factor.

Differences from [BY19]. In [BY19] section 12, the authors compute the matrix product in alg. 3 by recursively dividing it into halves, resulting in a total of $n-1$ matrix multiplications. They utilize this by keeping the precision of the entries as low as possible.

We compute the product differently, by instead computing it iteratively. Because we attempt to minimize the new code introduced to Fiat-Crypto, this requires $4n$ modular multiplications; and since only the top right entry of the final matrix is needed, it suffices to do matrix-vector multiplications (note that this is not possible when recursively dividing the product). However using this method, one cannot keep the precision low for as many multiplications. This was fine for our implementation, since keeping track of different precision (and using appropriate multiplication implementations) in Fiat-Crypto would be difficult. Our unverified implementation of the jumpdivstep approach keeps track of how these coefficients grow (one limb with every iteration of the outer loop), making it possible to delay the expensive modular reduction until it is strictly necessary (lazy reduction).

The half-deta optimization. We incorporate in our jumpdivstep implementation the faster variant proposed by Wuill *et al.* in [WMea21]. This variant starts with the value $\delta = 1/2$ and runs for around 18% fewer iterations, as given by the closed formula $\lfloor (45907 \log_2(M) + 26313)/19929 \rfloor$ for inversion modulo M. While the authors provide a mathematical correctness proof in the latest version of the repository for the result, we understand this has not passed peer-review and take the extra precaution with validating the lower number of iterations. We adapted the 256-bit Coq proofs in [WMea21] for our various parameter sizes and executed them with two optimizations: using the `native_compute` reduction machine in Coq, which cut execution time to 32 hours from the initially reported 2.5 days; and extracted the proofs using Coq's built-in extraction mechanism [Let02] to OCaml native binaries for another 2-factor reduction in time. Table 1 reports on our progress towards running all proofs. We expect finishing the remaining OCaml native proofs in about a month more of computation time.

Table 1: Our progress in running computational proofs to validate the lower number of iterations for various prime moduli sizes in the half-delta optimization using different proof strategies. A cell containing a number of hours indicates that the proof finished with a positive outcome in the specified time; or that it is still running otherwise.

Prime size (bits)	Iterations	Coq-`native_compute`	Coq-ExtractedOCaml
254	590	32.1 hours	14.7 hours
381	878	213 hours	100.5 hours
448	1033	In progress	281.1 hours
521	1201	In progress	In progress
575	1325	In progress	In progress

4.5 Experimental results

We have generated and benchmarked modular inversion in C for the primes commonly used in both ECC and PBC settings of curve-based cryptography. For the ECC case, we chose the well-known primes $2^{255} - 19$, $2^{448} - 2^{224} - 1$ and $2^{521} - 1$ labeled by their named curves Curve25519, Curve448 and NIST-P521 at the 128-, 224- and 256-bit security levels. For the PBC case, we took the base fields for Barreto-Naehrig (BN) [BN05] and Barreto-Lynn-Scott (BLS) curves [BLS02] at three different security levels. These are the 254-bit prime used in the now legacy 110-bit secure BN curves [AKL+11, MSS16], the 381-bit prime for BLS curves with embedding degree 12 undergoing standardization at 128-bit security [MSS16], and the 575-bit prime for BLS curves with embedding degree 48 proposed for 256-bit security [MAF20].

The generated code was integrated in the RELIC toolkit [AGM+], a cryptographic library containing several state-of-the-art implementations of pairings. RELIC uses a combination of hand-written Assembly (ASM) with higher-level C-code and has set speed records for several of the chosen parameters in the PBC case. Integrating the code with RELIC allowed convenient benchmarking to compare the efficiency of our approach with other field inversion algorithms already implemented in the library.

We present our results in Table 2 and Table 3. In both tables, the first part has baseline implementations from the GMP 6.2.1 library [Gt12] for reference. These timings

```
Definition outer_loop_body f g (v r : list Z) :=
  let '(_,f1,g1,u1,v1,q1,r1) :=
  fold_right (fun i data => twos_complement_word_full_divstep_aux machine_wordsize data)
             (1,nth_default 0 f 0,nth_default 0 g 0,1,0,0,1)
             (seq 0 (Z.to_nat (machine_wordsize - 2))) in
  dlet f2 := word_sat_mul machine_wordsize sat_limbs u1 f in
  dlet f3 := word_sat_mul machine_wordsize sat_limbs v1 g in
  dlet g2 := word_sat_mul machine_wordsize sat_limbs q1 f in
  dlet g3 := word_sat_mul machine_wordsize sat_limbs r1 g in
  dlet f4 := BYInv.sat_add machine_wordsize word_sat_mul_limbs f2 f3 in
  dlet g4 := BYInv.sat_add machine_wordsize word_sat_mul_limbs g2 g3 in
  dlet f5 := sat_arithmetic_shiftr machine_wordsize word_sat_mul_limbs f4 (machine_wordsize - 2) in
  dlet g5 := sat_arithmetic_shiftr machine_wordsize word_sat_mul_limbs g4 (machine_wordsize - 2) in
  dlet f6 := firstn sat_limbs f5 in
  dlet g6 := firstn sat_limbs g5 in
  dlet u2 := twos_complement_word_to_montgomery_no_encode u1 in
  dlet v02 := twos_complement_word_to_montgomery_no_encode v1 in
  dlet q2 := twos_complement_word_to_montgomery_no_encode q1 in
  dlet r02 := twos_complement_word_to_montgomery_no_encode r1 in
  dlet v2 := mulmod machine_wordsize n m m' u2 v in
  dlet v3 := mulmod machine_wordsize n m m' v02 r in
  dlet r2 := mulmod machine_wordsize n m m' q2 v in
  dlet r3 := mulmod machine_wordsize n m m' r02 r in
  dlet v4 := addmod machine_wordsize n m v2 v3 in
  dlet r4 := addmod machine_wordsize n m r2 r3 in
  (f6, g6, v4, r4).
```

Figure 3: Implementation of a jump-divstep in Fiat-Crypto

set a lower bound (aggressively optimized variable-time code) and upper bound (generic constant-time approach) that illustrate how challenging implementing field inversion in constant-time can be in terms of performance for general fields. The next part has timings for the FLT approaches using exponentiation by $p-2$. For the ECC primes, we took state-of-the-art timings from the literature in the ASM case [KN20, FLD19] and benchmarked the same addition chains over field arithmetic generated by Fiat-Crypto. For the PBC case, we built and benchmarked RELIC using both the existing ASM backends and field arithmetic code generated by Fiat-Crypto. These timings illustrate the penalty of going from handwritten field arithmetic in ASM to field arithmetic generated by Fiat-Crypto for the different parameters. The data indicates an approximate slowdown of a factor 1.2–1.7 when compiling using clang and a factor over 2 when using GCC. It is clear that GCC has much more trouble compiling Montgomery arithmetic generated by Fiat-Crypto.

The rest of tables benchmark describe the performance of various implementations of the Bernstein-Yang algorithm (alg. 2). The most interesting entries performance-wise are the jumpdivstep and hdjumpdivstep, which respectively are the jumpdivstep implementation that we generate automatically from Fiat-Crypto; and the variant using half-delta proposed recently. These are also benchmarked in RELIC in the PBC case using the provided ASM backends. The performance difference between the two ranges from a factor of 1.8 to 4.44, a reasonable trade-off considering the correctness guarantees provided by the Fiat-crypto version. More importantly, the hdjumpdivstep implementations outperform the FLT implementations in almost all cases, both for the ASM and Fiat-Crypto finite field arithmetic backends, except $2^{255}-19$. The speedups over FLT+Fiat increase for larger primes, illustrating that FLT approaches do not scale well for larger parameters.

Timings for Curve25519. We report detailed timings for the prime $2^{255}-19$ generated in the unsaturated representation. There are many applications for such an implementation, due to the widespread adoption of Curve25519 and Ed25519 as key exchange and digital signature algorithms [Ber06, BDL+12]. The Bernstein-Yang paper reports 8,778 Skylake cycles, later improved to 3,900 cycles[8]. An alternative implementation approach by Thomas Pornin [Por20] was benchmarked at 6,200 cycles in our Skylake processor.In comparison, the performance degradation of our best implementation is around 3.8 in comparison to those results, but we note that these faster implementations are not verified beyond exhaustive testing and/or they employ hand-written Assembly optimizations including vector instructions.

Table 2: Benchmarks of different approaches for field inversion over ECC fields on an Intel Skylake Core i7-6700K CPU running at 4.00GHz, using GCC version 11 and clang from LLVM 12. With the exception of *Variable-time GMP*, all operations are implemented in constant-time. Numbers were obtained by computing the average of 10^4 consecutive executions of an implementation measured using the cycle counter, and computing the average. TurboBoost and HyperThreading were disabled for benchmarking stability. Numbers in bold are the fastest for this work or related work among the different compilers for a certain prime.

			Curve25519		Curve448		NIST-P521	
	Verified	Auto	clang	gcc	clang	gcc	clang	gcc
Variable-time GMP	No	No	3,098	3,314	4,724	5,799	6,814	7,128
Constant-time GMP	No	No	75,895	76,300	186,637	186,186	257,935	270,085
FLT+ASM	No	No	–	**9,301**	*****41,400**	–	–	**53,828**
FLT+Fiat	No	Partially	13,638	18,253	52,344	54,957	82,679	61,327
This work+Fiat (divstep)	Yes	Yes	72,421	98,655	195,571	266,130	249,752	379,965
This work+Fiat (jumpdivstep)	Yes	Yes	17,696	19,531	48,144	53,677	65,832	76,740
This work+Fiat (hdjumpdivstep)	Yes	Yes	**14,837**	17,134	**39,541**	45,740	**52,828**	64,858

(*) The authors also report 35,000 for an AVX2 implementation, but we consider the 64-bit ASM implementation more fair for comparison.

Table 3: Benchmarks of different approaches for field inversion over pairing fields on an Intel Skylake Core i7-6700K CPU running at 4.00GHz, using GCC version 11 and clang from LLVM 12. With the exception of *Variable-time GMP*, all operations are implemented in constant-time. Numbers were obtained by computing the average of 10^4 consecutive executions of an implementation measured using the cycle counter, and computing the average. TurboBoost and HyperThreading were disabled for benchmarking stability. Numbers in bold are the fastest for this work or related work among the different compilers for a certain prime.

			BN-254		BLS12-381		BLS48-575	
	Verified	Auto	clang	gcc	clang	gcc	clang	gcc
Variable-time GMP	No	No	3,291	3,270	4,724	4,716	7,495	7,504
Constant-time GMP	No	No	75,639	76,168	146,157	146,083	270,631	271,168
FLT + ASM	No	No	**31,452**	31,492	104,513	**103,361**	288,719	**288,109**
FLT + Fiat	No	Partially	52,954	68,173	143,325	257,851	470,925	867,331
This work+ASM (divstep)	No	No	87,456	87,584	167,724	166,580	335,864	337,641
This work+ASM (jumpdivstep)	No	No	14,382	14,383	23,820	23,810	43,941	43,989
This work+ASM (hdjumpdivstep)	No	No	**9,777**	9,873	16,377	**16,183**	31,963	**27,911**
This work+Fiat (divstep)	Yes	Yes	79,653	118,305	181,344	289,786	403,309	655,831
This work+Fiat (jumpdivstep)	Yes	Yes	20,762	28,998	49,030	71,990	147,016	212,511
This work+Fiat (hdjumpdivstep)	Yes	Yes	**17,489**	23,394	**41,036**	62,018	**120,179**	183,996

[8]https://gcd.cr.yp.to/software.html

5 Formalization of Bernstein-Yang inversion

The main theorem that we have formalized is Theorem 1. A pervasive tactic used in our proofs is `lia` (linear integer arithmetic) and `lra` (linear real arithmetic). These tactics respectively solve (linear) inequalities over the integers and reals. It will use all available inequalities in the context, so a recurring method is to introduce all necessary inequalities to the context and then use `lia`/`lra` to solve the goal.

5.1 p-adic valuations

Since some theory of p-adic valuations were required for the proof we developed a small library for the basics of this theory. This is in the file `src/PadicVal.v`. We implementented the p-adic simply by counting the number of times a number is divisible by p and prove the following specification

```
Lemma pval_spec p a (Ha : a <> 0) (Hp : 1 < p) :
  (p ^+ (pval p a) | a) /\ ~ (p ^+ (S (pval p a)) | a).
```

i.e. that p to the power of `pval p a` divides a but not p to any higher power. We also prove uniqueness such that the full specification becomes

```
Lemma pval_full_spec p a v (Ha : a <> 0) (Hp : 1 < p) :
  pval p a = v <-> (p ^+ v | a) /\ ~ (p ^+ (S v) | a).
```

We also define split from and prove the specification

```
Lemma psplit_spec p a (Ha : a <> 0) (Hp : 1 < p) :
  a = (p ^+ (pval p a)) * psplit p a /\ ~ (p | psplit p a).
```

The proofs of these are all very straightforward. The proof of `pval_spec` is perhaps the most involved, but it follows by generalizing the amount of divisions and doing induction in that number.

5.2 The gcd algorithm

The formalization of the gcd algorithm related to divsteps (described in Appendix E in [BY19]) is in `src/AppendixE.v`. We prove three main things here

1. The existence and specification of $q(f, g)$ (Theorem E.1 in [BY19])
2. The correctness of the gcd algorithm assuming termination (Theorem E.3 in [BY19])

We implemented a function computing $q(f, g)$ instead of proving its existence abstractly. We did this to able to use it in the recursive definition of R_i (see Theorem E.2 in [BY19]). One minor issue here is that to construct $q(f, g)$ one needs to compute the inverse of $\mathrm{split}_2\, g$ as a 2-adic number. Recall that in the paper most theorems are stated over this larger ring. This inverse is not necessarily an integer (e.g. 2 does not have a multiplicative inverse), but when constructing $q(f, g)$ one only needs this inverse to a finite precision.

Thus, we implemented the function which computes the inverse of an integer in $\mathbb{Z}_2$ to a finite precision. This is implicit in our definition of $q(f, g)$ but to compute this, you compute the modular inverse of $\mathrm{split}_2\, g$ in 2^i to get the 2-adic inverse to precision i. This follows from the construction of $\mathbb{Z}_2$ as $\lim_{\leftarrow} \mathbb{Z}/2^i\mathbb{Z}$.

Another choice we made in this section was to use the real number type `R` defined in the Coq standard library which depends on the axioms of classical logic. This means that certain expressions no longer evaluate directly in Coq. We could have avoided this by developing a library of algebraic numbers. For example, the expression $f\, \mathrm{div}_2\, g$ is a rational number, but to avoid coercing it to a real number later on we define it as a term of `R`. Concretely, the definition is given as

```
Definition div_2 f g : R := IZR (q f g) / IZR (2 ^+ ord2 g).
```

where `IZR` is the embedding from the `Z` to `R` from the standard library (see also Theorem E.1 in [BY19]).

The formal proofs follows the paper quite closely. We use two custom induction principles in the proof of Theorem E.3, one for getting the induction hypothesis for the two previous naturals and one to make induction from $n > 0$ and down to zero. These are `induction2` and `rev_2_ind` from `src/InductionPrinciples.v`.

5.3 Complexity analysis

The formalization of the termination proof of the gcd algorithm is in `src/AppendixF.v`. This part of the proof is the content of Appendix F in [BY19]. Here, the proof becomes more complicated and there is one theorem from [BY19] which we do **not** formally prove in Coq due to performance constraints. We thus add it as an axiom in Coq.

The theorem is a computational one which is proven by observing that a certain program terminates. It was carried out in Sage [The20] by Bernstein and Yang (see theorem F.22 in [BY19]). We have implemented the algorithm in Coq's built-in functional language. When extracting the implementation to OCaml [Let02], it terminates in approx. 8 hours with the same output as reported in [BY19].

5.3.1 The operator norm

The first complication is how to define the operator norm of a matrix, since this is the measure we will use to prove termination. That is, we will prove that the operator norm of products of transition matrices is bounded by an exponentially decreasing sequence of numbers and then use this to prove termination of the gcd algorithm.

The standard definition of the operator norm as the supremum over lengths of images of unit vector under matrix-vector multiplication is however quite cumbersome since it requires quite a lot of analysis to prove that this supremum is computable. Computability is not necessary for Coq's classical reals, but we would like to use a smaller (computable) field in the future, for example the real algebraic numbers. Our solution is simply to only define operator norms for real 2 by 2 matrices and then use the associated formula. One major issue with this approach is that we lose all the nice analytical mathematical properties and have to do proofs algebraically. More concretely, the proofs of the properties $|Mv|_2 \leq |M||v|_2$ and $|MN|_2 \leq |M|_2|N|_2$ become a lot more involved.

In particular, the latter property is tedious to prove algebraically since the formula for the operator norm of a product becomes quite complex. To prove it we instead used the observation that is also used in the Bernstein-Yang paper to prove the formula of the operator norm *but* in the "opposite direction". This is the theorem F.11 in the paper. This proof however uses the spectral theorem for real matrices of 2 by 2 dimensions so we formalize this in `src/Spectral.v`. The proof of this is standard. We then use it in the proofs of the two operator norm properties in `mat_norm_vmult` and `mat_norm_mmult` in `src/Spectral.v`. See also the comment in `src/Spectral.v`.

The file `src/Spectral.v` only contains lemmas pertaining to the spectral theorem and the norm of matrices over reals. Our formalization of 2 by 2 matrices over general rings is in `Matrix.v`. This theory is built on top of a small theory of algebraic structures, which is the content of `src/Hierarchy.v`. The reason we developed our own small library instead of using an established one (such as the one in [MT20]) is to be able to do proofs by computation and have easier access to the implementation details (as these are abstracted away in e.g. [MT20]). We used automation in this development to avoid tedious algebraic proofs. The tactics `auto_mat` and `inversion_mat` use the decision procedure `ring` over types declared as algebraic rings to solve most equational proofs about matrix operations.

5.3.2 Bounding the operator norm

Next, the proof proceeds by computing a bound on the operator norm of products of matrices of a particular form, namely matrices given by the definition

```
Definition M (e : nat) (q : Z) := [ 0 , 1 / (2 ^ e) ; - 1 / (2 ^ e) , q / (2 ^ (2 * e)) ]
```

The bound is given by theorem F.16 in [BY19], which is formalized in `src/AppendixF.v`. The formalized proof is rather messy and ad hoc, since we did not find any published tactics for manipulating expressions involving square roots (`sqrt`). The general strategy is to reduce the expression to an expression without square roots by isolating and squaring appropriately. These two methods do not, however, generally suffice (consider e.g. $\sqrt{5} \leq 1 + \sqrt{2} + \sqrt{3}$).

We did consider developing a decision procedure to handle equations and inequalities involving square roots, but since the complexity of the expressions can grow exponentially unless one is careful we decided not to. It appears that developing such a procedure would be as involved as developing a type of integers extended with square roots, which would be much more interesting to pursue anyway (this could potentially also relieve us of having to use classical reals).

5.3.3 The bounding sequence

Next, we define the number sequence α_n which will bound the operator norms. Using this we prove two main facts

- If a particular subset of the matrices $M(e, q)$ are bounded by α_n, then all such matrices are bounded by α_n (Theorem F.21 in [BY19]).
- Prove that the gcd algorithm terminates (Theorem F.26 in [BY19]).

Now, notably we **do not** formally prove Theorem F.22 in [BY19], i.e. that the matrices on this particular subset are bounded by α_n. This is proof that we discussed at the beginning of Section 5.3.

The other proofs in this section were more straightforward as they mostly combine previously established theorems (about matrices and about the R_j sequence from `src/AppendixE.v`).

We developed a small "big operation" library to reason about the big multiplications and big additions, as required in theorem F.21. Similarly to our theory of matrices, it is built on top of `Hierarchy.v`. Again, as with the reasoning for using our own matrix library, this was to have a greater degree of control over the implementation.

5.4 Relating the gcd algorithm and divstep

We finally prove Theorem 1 in `Section11.v` by proving and utilizing the connection between the gcd algorithm and iterating divstep in `src/AppendixG.v`. We have also included an implementation of the inversion algorithm, alg. 2, in `src/BYInv.v` with an accompanying correctness proof.

The proofs in `src/AppendixG.v` did not introduce too many complications since they are mostly about combining the previous results. There is however again one theorem here that we **do not** prove. This is theorem G.4 from [BY19]. Again this is a computational proof which proves difficult to execute inside Coq. The algorithm that has to be executed does however not require any particularly advanced operations, so we have succeeded in extracting a version from Coq to OCaml which can be run. This is implemented in `src/Comp2/Definitions.v` and `src/Comp2/Extraction.v`. To achieve a high enough performance we use native 63-bit integers instead of the regular arbitrary precision integers implemented in Coq.

The content of this theorem is basically a bound on number of iteration of divstep required before termination for small values (by brute force). Thus if one is not interested in computing modular inverses for small numbers, then this lemma is not even necessary for correctness. More concretely, if the modulus has more than 21 bits, then theorem G.4 is not necessary (see theorem G.6). Since this will almost always be the case for cryptographic purposes we consider the lack of proof less critical.

6 Related work

The verified *synthesis* approach adopted by Fiat-Crypto is not the only possibility for verifying implementations of cryptography algorithms. An alternate approach is writing optimized code by hand in a low level language embedded in a proof assistant.

EverCrypt [PPF+20] is one example of this approach that provides a formally verified cryptographic provider, i.e., a collection of verified cryptographic implementation together with an API. It builds on two other projects HACL* [ZBPB17], which is a collection of cryptographic protocols implemented, specified and verified in a subset of the the F* language (Low*) and compiled to C, and ValeCrypt which is a collection of cryptographic primitives implemented in an assembly language and verified using the Vale tool [BHK+17]. EverCrypt does not generate code for new primitives, but it supports a large amount of cryptographic primitives including AES, SHA-3, MD5 and implementations of elliptic curves as well as signature and key exchange protocols on top of these. Notably, EverCrypt does not support curves for pairing-based cryptography. Yet another approach is followed by Jasmin [ABB+17], which provides a higher level assembly language and a verified compiler to write Intel Assembly directly. Jasmin has been used to verify a high-performance SHA3 implementation [ABB+19].

There are differences in the guarantees of the tools mentioned. Coq is *foundational* in that it reduces all proofs to the axioms of mathematics. Neither Easycrypt nor F* are foundational, even though they were carefully designed, they dependent on an unverified reduction to unverified SMT-solvers.

7 Future Work

We can extend our work in several directions. In one angle, we can target embedded platforms running over ARM or RISC-V and study the performance trade-offs in those systems. In another angle, we can extend the scope to other arithmetic layers employed in many other cryptographic protocols based on pairings or isogenies. Fiat-Crypto can be extended with the general construction of field extension, by implementing polynomial arithmetic. Since these polynomials would have coefficients in the finite fields currently generated by Fiat-Crypto, one would have to be able to generate representations of these, e.g. as arrays of integers. It is unclear whether or not Fiat is geared for this, but if it is then the implementation should be no more difficult than what is presented in Section 4.

Going even further, one could extend Fiat-Crypto to generate elliptic curve arithmetic directly. One obstacle is that Fiat-Crypto's low-level language does not include function calls which are necessary to implement elliptic curve operations. One prospective solution is to use the Rupicola compiler, which has some links to the Fiat-Crypto library. Rupicola is a verified compiler from a subset of Coq to bedrock2, which is a low-level language embedded in Coq. One can then go from bedrock2 to C or RISC-V to obtain an efficient implementation.

After generating elliptic curve arithmetic, we can go yet another step up the abstraction layer and implement billinear pairings over these curves. Bilinear pairings are maps from a product of curve groups of prime order to the multiplicative subgroup of an extension field. At this point we would have all necessary primitives to implement pairing-based protocols. This has the added complexity of requiring a formalization of bilinear pairings in Coq. Formalizations of elliptic curves in Coq already exist, e.g. in [BS14] and in the Fiat-Crypto library, but no formalized implementation of a pairing has been developed.

References

[ABB+17] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. Jasmin: High-assurance and high-speed cryptography. In *CCS*, pages 1807–1823. ACM, 2017.

[ABB+19] José Bacelar Almeida, Cecile Baritel-Ruet, Manuel Barbosa, Gilles Barthe, François Dupressoir, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Alley Stoughton, and Pierre-Yves Strub. Machine-checked proofs for cryptographic standards: Indifferentiability of sponge and secure high-assurance implementations of SHA-3. In *CCS*, pages 1607–1622. ACM, 2019.

[ABLR13] Diego F. Aranha, Paulo S. L. M. Barreto, Patrick Longa, and Jefferson E. Ricardini. The Realm of the Pairings. In *Selected Areas in Cryptography*, volume 8282 of *LNCS*, pages 3–25. Springer, 2013.

[AFK+12] Diego F. Aranha, Laura Fuentes-Castañeda, Edward Knapp, Alfred Menezes, and Francisco Rodríguez-Henríquez. Implementing Pairings at the 192-Bit Security Level. In *Pairing*, volume 7708 of *LNCS*, pages 177–195. Springer, 2012.

[AGB20] Alejandro Cabrera Aldaya, Cesar Pereida García, and Billy Bob Brumley. From A to Z: projective coordinates leakage in the wild. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(3):428–453, 2020.

[AGM+] D. F. Aranha, C. P. L. Gouvêa, T. Markmann, R. S. Wahby, and K. Liao. RELIC is an Efficient LIbrary for Cryptography. `https://github.com/relic-toolkit/relic`.

[AGTB18] Alejandro Cabrera Aldaya, Cesar Pereida García, Luis Manuel Alvarez Tapia, and Billy Bob Brumley. Cache-timing attacks on RSA key generation. *IACR Cryptology ePrint*, 2018:367, 2018.

[AKL+11] Diego F. Aranha, Koray Karabina, Patrick Longa, Catherine H. Gebotys, and Julio López. Faster Explicit Formulas for Computing Pairings over Ordinary Curves. In *EUROCRYPT*, volume 6632 of *LNCS*, pages 48–68. Springer, 2011.

[Ara17] D. F. Aranha. Pairings are not dead, just resting. `https://ecc2017.cs.ru.nl/slides/ecc2017-aranha.pdf`, 2017.

[ASS17] Alejandro Cabrera Aldaya, Alejandro Cabrera Sarmiento, and Santiago Sánchez-Solano. SPA vulnerabilities of the binary extended euclidean algorithm. *J. Cryptogr. Eng.*, 7(4):273–285, 2017.

[BBPV12] Billy Bob Brumley, Manuel Barbosa, Dan Page, and Frederik Vercauteren. Practical realisation and elimination of an ecc-related software bug attack. In *CT-RSA*, volume 7178 of *LNCS*, pages 171–186. Springer, 2012.

[BCG$^+$14] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized Anonymous Payments from Bitcoin. In *S&P'14*, pages 459–474. IEEE Computer Society, 2014.

[BD17] Razvan Barbulescu and Sylvain Duquesne. Updating key size estimations for pairings. *IACR Cryptology ePrint*, 2017:334, 2017.

[BDL$^+$12] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *J. Cryptogr. Eng.*, 2(2):77–89, 2012.

[Ber06] Daniel J. Bernstein. Curve25519: New diffie-hellman speed records. In *Public Key Cryptography*, volume 3958 of *Lecture Notes in Computer Science*, pages 207–228. Springer, 2006.

[BHK$^+$17] Barry Bond, Chris Hawblitzel, Manos Kapritsos, K. Rustan M. Leino, Jacob R. Lorch, Bryan Parno, Ashay Rane, Srinath T. V. Setty, and Laure Thompson. Vale: Verifying high-performance cryptographic assembly code. In *USENIX Security Symposium*, pages 917–934. USENIX Association, 2017.

[BLS02] Paulo S. L. M. Barreto, Ben Lynn, and Michael Scott. Constructing elliptic curves with prescribed embedding degrees. In *SCN*, volume 2576 of *LNCS*, pages 257–267. Springer, 2002.

[BLS04] Dan Boneh, Ben Lynn, and Hovav Shacham. Short Signatures from the Weil Pairing. *J. Cryptology*, 17(4):297–319, 2004.

[BN05] Paulo S. L. M. Barreto and Michael Naehrig. Pairing-Friendly Elliptic Curves of Prime Order. In *Selected Areas in Cryptography*, volume 3897 of *LNCS*, pages 319–331. Springer, 2005.

[Bos14] Joppe W. Bos. Constant time modular inversion. *J. Cryptogr. Eng.*, 4(4):275–281, 2014.

[BS14] Evmorfia-Iro Bartzia and Pierre-Yves Strub. A formal library for elliptic curves in the Coq proof assistant. In *ITP*, volume 8558 of *LNCS*, pages 77–92. Springer, 2014.

[BY19] Daniel J. Bernstein and Bo-Yin Yang. Fast constant-time gcd computation and modular inversion. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019(3):340–398, 2019.

[EPG$^+$19] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. Simple high-level code for cryptographic arithmetic - with proofs, without compromises. In *IEEE Symposium on Security and Privacy*, pages 1202–1219. IEEE, 2019.

[FLD19] Armando Faz-Hernández, Julio César López-Hernández, and Ricardo Dahab. High-performance implementation of elliptic curve cryptography using vector instructions. *ACM Trans. Math. Softw.*, 45(3):25:1–25:35, 2019.

[Gt12] Torbjörn Granlund and the GMP development team. *GNU MP: The GNU Multiple Precision Arithmetic Library*, 6.2.1 edition, 2012. https://gmplib.org/.

[KN20] Palash Sarkar Kaushik Nath. Efficient arithmetic in (pseudo-)mersenne prime order fields. *Advances in Mathematics of Communications*, 0, 2020. To appear.

[Let02] Pierre Letouzey. A new extraction for coq. In *TYPES*, volume 2646 of *Lecture Notes in Computer Science*, pages 200–219. Springer, 2002.

[lfS17] luigi1111 and Riccardo "fluffypony" Spagni. Disclosure of a major bug in cryptonote based currencies, 2017. URL `https://web.getmonero.org/2017/05/17/disclosure-of-a-major-bug-in-cryptonote-based-currencies.html`.

[MAF20] Narcisse Bang Mbang, Diego F. Aranha, and Emmanuel Fouotsa. Computing the optimal ate pairing over elliptic curves with embedding degrees 54 and 48 at the 256-bit security level. *Int. J. Appl. Cryptogr.*, 4(1):45–59, 2020.

[Mon85] Peter L Montgomery. Modular multiplication without trial division. *Mathematics of computation*, 44(170):519–521, 1985.

[MSS16] Alfred Menezes, Palash Sarkar, and Shashank Singh. Challenges with assessing the impact of NFS advances on the security of pairing-based cryptography. In *Mycrypt*, volume 10311 of *LNCS*, pages 83–108. Springer, 2016.

[MT20] Assia Mahboubi and Enrico Tassi. *Mathematical Components*. Zenodo, November 2020. doi:10.5281/zenodo.4282710.

[Por20] Thomas Pornin. Optimized binary gcd for modular inversion. Cryptology ePrint 2020/972, 2020.

[PPF$^+$20] Jonathan Protzenko, Bryan Parno, Aymeric Fromherz, Chris Hawblitzel, Marina Polubelova, Karthikeyan Bhargavan, Benjamin Beurdouche, Joonwon Choi, Antoine Delignat-Lavaud, Cédric Fournet, Natalia Kulatova, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy, Christoph M. Wintersteiger, and Santiago Zanella Béguelin. Evercrypt: A fast, verified, cross-platform cryptographic provider. In *IEEE Symposium on Security and Privacy*, pages 983–1002. IEEE, 2020.

[Sol99] Jerome A. Solinas. Generalized Mersenne numbers. Technical report, 1999.

[Tea20] The Coq Development Team. The coq proof assistant, version 8.12.0, July 2020. doi:10.5281/zenodo.4021912.

[The20] The Sage Developers. *SageMath, the Sage Mathematics Software System (Version 8.9.0)*, 2020. `https://www.sagemath.org`.

[WMea21] Pieter Wuille, Gregory Maxwell, and Russell O'Connor et al. Bounds on divsteps iterations in safegcd. `https://github.com/sipa/safegcd-bounds`, Accessed in April 2021.

[ZBPB17] Jean Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. Hacl*: A verified modern cryptographic library. In *CCS*, pages 1789–1806. ACM, 2017.