High-assurance field inversion for curve-based cryptography

Benjamin Salling Hvass
Department of Computer Science
Aarhus University, Denmark
bsh@cs.au.dk

Diego F. Aranha
Department of Computer Science
Aarhus University, Denmark
dfaranha@cs.au.dk

Bas Spitters
Department of Computer Science
Aarhus University, Denmark
spitters@cs.au.dk

Abstract—**The security of modern cryptography depends on multiple factors, from sound hardness assumptions to correct 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 a cryptographic computation, with a small impact on 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 [1]–[3]. In this work, we focus on implementing field inversion for primes of cryptographic interest with security against timing attacks, irrespective of whether the FLT-based inversion can be efficiently implemented. We extend the Fiat-Crypto framework, which synthesizes provably correct-by-construction implementations, to implement the Bernstein-Yang inversion algorithm as a step towards this goal. This allows a correct implementation of prime field inversion to be 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 for the largest parameters. Our work is already used in production in the MirageOS unikernel operating system, `zig` programming language, and the ECCKiila framework [4].**

Index Terms—**Field arithmetic, Constant-time execution, Implementation security, Formal verification**

I. 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 efficiently and securely 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 high performance penalty [5]. For this reason, field inversion is often implemented using Fermat's Little Theorem (FLT) approach of exponentiating by $p-2$ in $\mathbb{F}_p$ for prime p. This is efficient for ECC implementations relying on special primes with fast modular reduction, especially when the exponent allows a short addition chain as in Curve25519 [6]. When performance is more pressing or parameters are not friendly to FLT inversion, implementers typically resort to an aggressively optimized version of the Extended Euclidean Algorithm (EEA). However, bugs and side-channel leakage in the EEA implementation can lead to attacks against RSA [2] and ECC [1], [3]. These are not just threats of research interest, as illustrated by a vulnerability recently discovered in the EEA implementation in Windows that could be exploited to mount denial of service attacks[1].

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 families of pairing-friendly of curves [7], [8] that employ the slower modular reduction in Montgomery arithmetic [9]. In the context of PBC, the performance of field inversion matters during exponentiation in pairing groups, and it also unlocks an optimization called compressed squarings in the final exponentiation of the pairing [10]. With pairings being increasingly deployed as a fundamental building block for zero-knowledge proofs and privacy-preserving cryptocurrencies (for example in short signature schemes [11] and zkSNARKs [12]), 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 [13][2]. A survey of implementation bugs in cryptographic libraries is collected in [14], [15].

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 [16], [17]. 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

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

[2] https://web.getmonero.org/2017/05/17/disclosure-of-a-major-bug-in-cryptonote-based-currencies.html

using the popular BLS12-381 curve for motivation, the cost of one scalar multiplication required for computing short Boneh-Lynn-Schacham signatures [11] is reported to be around 400,000 cycles on Intel Skylake [18], [19] using variable-time inversion. According to our benchmarking in Table III, 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 significant and motivates the need for our 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 [20]. In that same year, an alternative path for implementing cryptographic libraries was demonstrated as viable in the Fiat-Crypto framework [14]. 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. Simultaneously, the generated code stays within a small imperative language, thus avoiding notorious memory safety issues which cause many vulnerabilities [15]. 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-delta" variant, recently developed to optimize inversion within ECDSA signing over the curve `secp256k1` adopted in Bitcoin [21]. This variant requires a lower number of division steps to be evaluated, which immediately translates to better performance.

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 portable implementation of the algorithm for *any* prime using the two main representations supported in Fiat-Crypto (unsaturated Solinas and Montgomery). This comes in contrast with previous work, which consisted of implementing the FLT approach on top of a verified multiplier, instead of a dedicated specialized inversion algorithm [22]. Our formulation of the algorithm maximally relies on what is provided by Fiat-Crypto, taking advantage of the field operations provided by the framework whenever possible instead of introducing new ones. In the context of Montgomery arithmetic, this introduces some expensive multiplications to update the algorithm's 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 5.3 in comparison 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. The slowdown is 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 unverified assembly, with speedups ranging from 1.6 to 2 for different sizes. For the ECC primes, we outperform the FLT approaches in the two largest parameters and improve performance up to 40% against an implementation based on Fiat-Crypto and 14% against handwritten assembly.

Our slowest implementation is already used in production the MirageOS unikernel[3] and `zig` language [4] and the ECCKiila framework [4], showing that it is fast enough for engineering projects with a focus on correctness.

Outline of the paper. We briefly explain the necessary preliminaries of Fiat-Crypto and the inversion algorithm in Sections II and III. Sections IV to VI describe our implementation/formalization of the algorithm and our formalization of the correctness proof, respectively. The two final sections conclude with related and future work.

II. The Fiat-Crypto Framework

Fiat Cryptography [14] (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[5], BoringSSL[6] and the WireGuard VPN[7].

In Fiat-Crypto there are separate binaries to generate code for each style of multi-precision arithmetic: Montgomery, saturated and unsaturated representations. Although, there is no formal proof that the code is of constant time, only "straight-line code" is generated, i.e. code without branching that should run in constant-time after it is processed by an optimizing compiler. Fiat-Crypto consists of a verified compiler written in the Coq proof assistant [23]. It compiles from a subset of Coq to a simple language embedded in Coq containing only bitwise and machine-integer operations. From here, the generated terms can be pretty-printed to the programming languages mentioned above.

[3] https://github.com/mirage/mirage-crypto/tree/main/ec/native

[4] https://github.com/ziglang/zig/blob/master/lib/std/crypto/pcurves/common.zig

[5] https://blog.mozilla.org/security/2020/07/06/performance-improvements-via-formally-verified-cryptography-in-firefox/

[6] https://boringssl.googlesource.com/boringssl/+/master/third_party/fiat/

[7] https://www.wireguard.com/formal-verification/

Correct-by-construction vs. verifying existing code. Fiat-Crypto differs significantly from other verification projects: 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 multi-language support: in general each implementation in a different language would present a separate formalization effort.

Coq. Coq is a state-of-the-art interactive proof assistant based on dependent type theory [23]. 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 built-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 and it is finally checked by the kernel for correctness.

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

Multi-precision arithmetic. In cryptography, it is common to compute 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 the 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 `Z`. Note that the representation is little-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 computing on such a 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 approaches, 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 [24]), 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. The efficiency depends on the coefficients and exponents of the integral polynomial. 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 [6].

Montgomery Reduction. In addition to Generalized Mersenne Reduction, Fiat-Crypto supports Montgomery arithmetic [9]. 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 can be chosen as a power of 2 such that divisions become cheap and 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:

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

Because Montgomery reduction has the same complexity as 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.

III. Bernstein-Yang inversion

The Bernstein-Yang (BY) inversion algorithm [20] is a new and efficient constant-time algorithm for inverting in finite fields. In this paper we will only be using the algorithm over a field $\mathbb{F}_p$, for prime p. The algorithm is a constant-time variant of the classical Extended Euclidean Algorithm (EEA). We implement the BY algorithm in Fiat-Crypto (Section IV), and formalize its proof of correctness (Section V).

A. Specification and correctness

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

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

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 the divstep function.

We will also use the following *transition matrices*

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

These are transition matrices in the sense that multiplication corresponds to applying divstep once (up to a factor; see also Theorem 9.1 in [20]). Note that this definition differs slightly from the one in [20] (it is scaled by a factor).

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 depicted in Algorithm 1.

Note that while Algorithm 1 as described is not constant time, the branch can be implemented as a conditional swap (which can be implemented in constant time). This is also how it is implemented in [20].

For integers δ, f and g we write $(\delta_n, f_n, g_n) = \operatorname{divstep}^n(\delta, f, g)$ and $\mathcal{T}_n = \mathcal{T}(\delta_n, f_n, g_n)$.

```
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 $\mathcal{T}_n \mathcal{T}_{n-1} \cdots \mathcal{T}_0$
  $u \leftarrow 1,\ v \leftarrow 0,\ q \leftarrow 0,\ r \leftarrow 1$ ;
  for $i \leftarrow 1$ to $n$ do
      if $0 < \delta$ and $g$ odd then
          $\delta \leftarrow -\delta,\ f \leftarrow g,\ g \leftarrow -f,\ u \leftarrow q,\ v \leftarrow r,$
            $q \leftarrow -u,\ r \leftarrow -v$ ;
      $g_0 \leftarrow g \bmod 2$ ;
      $\delta \leftarrow \delta + 1$ ;
      $g \leftarrow \frac{g + g_0 f}{2}$ ;
      $q \leftarrow q + g_0 u$ ;
      $r \leftarrow r + g_0 v$ ;
      $u \leftarrow 2u$ ;
      $v \leftarrow 2v$ ;
  return $\delta, f, g, \left(\begin{smallmatrix} u & v \\ q & r \end{smallmatrix}\right)$
```

The DIVSTEPS procedure can then be used to implement modular inversion as described in Algorithm 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.

```
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 $\delta, f, g, \left(\begin{smallmatrix} u & v \\ q & r \end{smallmatrix}\right) \leftarrow$ DIVSTEPS$(m, \delta, f, g)$ ;
9 $g^{-1} \leftarrow e \cdot v \cdot \operatorname{sgn}(f)$ ;
```

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

Theorem 1 (Theorem 11.2 in [20]). *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 $(\delta_i, f_i, g_i) = \operatorname{divstep}^i(1, f, g)$ 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 Algorithm 2 follows from Theorem 1 since f and g are assumed to be coprime, the final values of f and v are respectively f_m and v_m, and p is the inverse of 2^m modulo f, so the following holds:

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

The theorem as stated here differs slightly from the one in [20] since our definition of $\mathcal{T}$ is scaled by a factor to avoid having to reason about rational numbers.

B. Outline of proof

The proof of Theorem 1, as given in [20], is in 4 parts:

- Specification of a related algorithm for computing the gcd of two numbers.
- Complexity analysis of the related algorithm; in particular giving a worst-case bound.
- Establishing the relation between divsteps and the related algorithm.
- Proving that reaching a fixed point of divstep yields the modular inverse.

These are described in Appendix E, F, G and Section 11 in [20], respectively.

We will expand on how each part was formalized in Section V. 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 $\operatorname{ord}_p g$ or $\operatorname{val}_p g$ (in the literature ν_p is also common). We will also write $\operatorname{split}_p g$ for g divided by this maximal power of p, i.e. $\operatorname{split}_p g = g/p^{\operatorname{ord}_p g}$.

While the proof in the paper uses 2-adic integers, we only use the corresponding statements for integers. This facilitates the formalization and suffices to prove Theorem 1.

C. 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 working on smaller numbers and "jumping" through the DIVSTEPS computations in larger steps. This optimized version is depicted in Algorithm 3 (see section 10 in [20] for details).

```
Algorithm 3: JUMPDIVSTEPS
  Input  : Integers $n, k, b, \delta, f$ and $g$ such that $f$ is odd
           and $k \mid n$ and $k \leq b$
  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 $\mathcal{T} \leftarrow \left(\begin{smallmatrix}1&0\\0&1\end{smallmatrix}\right)$ ;
2 for $i \leftarrow 1$ to $n/k$ do
3     $f' \leftarrow f \bmod 2^b, g' \leftarrow g \bmod 2^b$ ;
4     $\delta, f', g', \mathcal{T}' \leftarrow$ DIVSTEPS$(k, \delta, f', g')$ ;
5     $\left(\begin{smallmatrix}f\\g\end{smallmatrix}\right) \leftarrow \mathcal{T}\left(\begin{smallmatrix}f\\g\end{smallmatrix}\right)/2^k$ ;
6     $\mathcal{T} \leftarrow \mathcal{T}' \cdot \mathcal{T}$ ;
7 return $\delta, f, g, \mathcal{T}$
```

One way to see that Algorithm 3 is correct is to note that one run through the loop corresponds to k runs through the loop in Algorithm 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 $k-1$ first bits in $(g-f)/2$ and $(g + (g \bmod 2)f)/2$ (the two possibilities for subsequent g-values) 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 Algorithm 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 $i \leq j$,

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

This 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 [20]). 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 ith iteration).

IV. VERIFIED AND EFFICIENT FIELD INVERSION IN FIAT-CRYPTO

We will now shift gears and focus on the technical details of our formalization. Our contributions can be found in our fork of Fiat-Crypto at https://github.com/bshvass/fiat-crypto/tree/main. The generated code for the programming language `lang` can be found in the 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 folder.

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 V. The majority of our contribution to Fiat-Crypto is in the `src/Arithmetic/BYInv` folder and in the present section all file paths will be relative to this root.

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 Algorithm 1, since this was absent from the framework. In the following, `machine_wordsize : Z` or `mw : Z` will refer to the machine word size for which the implementation is parameterized. We will use `m : Z` to refer to the prime underlying the modular arithmetic and wrt. which we are trying to invert; `f g : list Z` will usually refer to bound multi-limb numbers and `a b : Z` to word-sized integers. The function `eval : list Z -> Z` will refer to evaluation of multi-limb numbers wrt. some weight function. In the context of Montgomery arithmetic this will be a uniform (or saturated) weight function and in the context of unsaturated Solinas it will be unsaturated.

When programming in Fiat-Crypto, one has to use the supported low-level language, i.e. the language whose terms can be compiled into the embedded C-like language and consequently generate C 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 n` which interprets a number to be of bit-length n and then flips all bits in its binary representation (bitwise negation).

A. Representing signed word-sized integers

We use the following definition to represent the numbers from -2^{mw} to $2^{\mathrm{mw}} - 1$

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

Then, e.g. `twos_complement mw (2 ^ mw - 1) = -1` as usual for a two's-complement representation. We will usually shorten `twos_complement` to `tc` and omit the bitwidth if it is implicit. We implement several operations working on these representations. They are all found at the bottom of the file `src/Util/ZUtil/Definitions.v` and have separate files in the same folder for their properties. In particular, we provide implementations of arithmetic shifts, addition in two's complement and multiplication in two's complement. These were all needed for the implementation, for the DIVSTEPS over words in JUMPDIVSTEPS. The correctness statements about these implementations prove how interpreting their results in two's complement corresponds to operations on their inputs in two's complement. For example, the correctness of shifting can be expressed by the lemma below.

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

B. *Representing signed multi-limb integers*

We extend Fiat-Crypto with signed multi-limb arithmetic. These numbers will be represented using lists of integers and evaluated as a multi-limb number in saturated representation interpreted in two's complement. In Coq this is defined as

```
Definition tc_eval (n : nat) (f : list Z) :=
  tc (mw * n) (eval n f).
```

where `eval` is evaluation of multi-limb numbeers in saturated representation (wrt. `mw`). The variable `n` is the amount of limbs needed, i.e. the length of f, and the evaluation is interpreted in two's complement at bitwidth `bw * n`. We will usually omit the `mw` and `n` parameters for brevity. All operations we support for multi-limb signed arithmetic can be found in `Definitions.v`. We will summarize a few of them here.

Arithmetic right shift. As for word-sized integers, we will need to divide multi-limb numbers by powers of two. We do this in the obvious way by shifting each limb and xor'ing the underflow with the shift of the next limb. Additionally, the most significant limb has to be arithmetically shifted to preserve the sign (here we can reuse our implementation for word-sized integers). The function `arithmetic_shiftr` does not fit Fiat-Crypto's DSL (since shifting by a non-constant is not supported), so we cannot synthesize its implementation for general k. However, we can do so for each specific k; we only need to instantiate k before generating. In the case of jumpdivstep, we instantiate k to $b-2$, where b is the machine word length. The correctness theorem is as follows

```
Lemma tc_eval_arithmetic_shiftr (f : list Z) (k : Z) (...) :
  tc_eval (arithmetic_shiftr f k)
    = tc_eval f / (2 ^ k).
```

where k is the shift amount and f is a multi-limb integer.

Addition, negation and subtraction. Addition is already implemented in Fiat-Crypto (signed addition is the same as unsigned addition in two's complement), so we simply wrap this implementation in the function `tc_add`. To negate we flip all bits of all limbs in the list and then use `tc_add` to add one. Subtraction is defined by combining addition and negation.

C. *Implementing* DIVSTEPS

We provide implementations of a single divstep, i.e. the body of the for loop in Algorithm 1.

```
Definition divstep_aux data :=
  let '(d,f,g,v,r) := data in
  let cond := land (pos d) (mod2 g) in
  let d' := zselect cond d (opp d) in
  let f' := select cond f g in
  let g' := select cond g (tc_opp f) in
  let v' := select cond v r in
  let v'':= addmod v' v' in
  let r' := select cond r (oppmod v) in
  let g0 := mod2 g' in
  let d'' := (d' + 1) mod 2 ^ mw in
  let f'' := select g0 tc_zero f' in
  let g'' := arithmetic_shiftr1 (tc_add g' f'') in
  let v''' := select g0 mont_zero v' in
  let r'' := addmod r' v''' in
 (d'',f',g'',v'',r'').

Definition divstep (d : Z) (f g v r : list Z) :=
  divstep_aux (d, f, g, v, r).
```

Listing 1: Implementation of a divstep in Fiat-Crypto

The implementations are in the file `Definitions.v` and properties about the implementations are proven in `Divstep.v`. There are implementations for Montgomery arithmetic, unsaturated Solinas and for word-sized divsteps (needed for JUMPDIVSTEPS). We have included the implementation using Montgomery arithmetic in Listing 1.

The implementation uses modular arithmetic for variables u, v, q and r 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 utilize the `zselect` and `select` functions provided by the library, which implement constant-time selection of values depending on a condition. The functions `addmod` and `oppmod` are multi-limb modular addition and negation, respectively; these were also implemented in Fiat-Crypto. The functions which we have implemented and verified are `pos`, which checks positivity of a word-sized integer in two's complement, `mod2`, which checks the parity of a multi-limb integer, and `arithmetic_shiftr1` which is a singular arithmetic right shift of multi-limb numbers. We did not have to implement `tc_add`, but we had to specify how it is computed in two's complement. Note also that the implementation differs slightly from the specification in Algorithm 1, in that we do not compute the u and q quantities. This is simply because they are not necessary in Algorithm 2 (they are however used in the proof of correctness of Algorithm 2).

We prove that executing `divstep` and then evaluating corresponds to the reference Coq function

```
Definition divstep_vr_mod m '(d, f, g, v, r) :=
  if Z.odd g
  then if 0 <? d
       then (1 - d, g, (g - f) / 2,
            (2 * r) mod m, (r - v) mod m)
       else (1 + d, f, (g + f) / 2,
            (2 * v) mod m, (r + v) mod m)
  else (1 + d, f, g / 2,
       (2 * v) mod m, r mod m).
```

which is simply defined over the `Z` type of integers in Coq, i.e. it does not compute over lists of numbers.

Concretely, we prove the theorem

```
Lemma divstep_correct (d : Z) (f g v r : list Z) (...) :
let '(d1,f1,g1,v1,r1) := (divstep_aux (d, f, g, v, r)) in
(tc d1, tc_eval f1, tc_eval g1,
       eval v1 mod m, eval r1 mod m) =
  divstep_vr_mod m (tc d, tc_eval f, tc_eval g,
                          eval v mod m, eval r mod m).
```

which states that for a given input d, f, g, v, r to divstep computing the Fiat-Crypto implementation Listing 1 and interpreting the output in two's complement, corresponds to interpreting d in two's complement, f, g as multi-limb numbers in two's complement, and r, v as multi-limb numbers modulo m, and computing `divstep_vr_mod_m` of the interpretations.

The proof is straightforward, in that we simply have to propagate the correctness theorems of each function that is called (`addmod`, `tc_add`, etc.). The correctness of the modular operations are provided by Fiat-Crypto, but others we have to prove ourselves. Most functions have a separate file dedicated to their correctness and properties, e.g. `TCAdd.v`, `ArithmeticShiftr.v`, `Mod2.v`, etc.

Note that the theorem has several assumptions not depicted here. Most notably, f and g must be less than half the maximum integer, since otherwise the first `tc_add` might overflow. Also, f have to be assumed odd, but recall that this is simply by the definition of divstep (see Section III).

D. *Implementing* BY-INVERSION *using* DIVSTEPS

We provide implementations of Algorithm 2 using DIVSTEPS. The implementations are in the file `Definitions.v` and properties about the implementations are proven in `BYInv.v`. There are implementations for Montgomery arithmetic and unsaturated Solinas. We have included the implementation using Montgomery arithmetic here in Listing 3.

In the implementation, we use the function `partition`, which simply takes an integer and represents it as a multi-limb number (as a list) according to some weight function. Obviously, the arguments to this has to be known a compile time and is a way to get Coq to compute constants and translate them to multi-limb numbers. We use it to get the representation of m, the prime wrt. which we are inverting, and `divstep_precomp`, which is the e from Algorithm 2. The fold in Listing 3 computes the for loop from Algorithm 2 and the function `iterations` computes

```
Definition iterations (b : Z) :=
  if b <? 46 then (49 * b + 80) / 17
    else (49 * b + 57) / 17.
```

as expected by Algorithm 2. Note that `iterations` is in `Ref.v`.

```
Theorem eval_by_inv (g : list Z)
  (g_length : length g = tc_limbs)
  (g_in_bounded : in_bounded g)
  (g_bounds :
    - 2 ^ (mw * tc_limbs - 2) <
      tc_eval g <  2 ^ (mw * tc_limbs - 2)) :
  eval (by_inv g) mod m = by_inv_ref m (tc_eval g).
```

Listing 2: Correctness of `by_inv`

```
Definition by_inv (g : list Z) :=
  let bits := (Z.log2 m) + 1 in
  let msat := partition m in
  let its := iterations bits in
  let pc := partition divstep_precomp in
  let '(_, fm, _, vm, _) :=
    fold_left (fun data i => divstep_aux data)
              (seq 0 (Z.to_nat its))
              (1,msat,g,zero,one) in
  let sign := tc_sign_bit fm in
  let inv := mulmod pc vm in
  let inv := select sign inv (oppmod inv) in
  inv.
```

Listing 3: Implementation of a BY-INVERSION in Fiat-Crypto using Montgomery arithmetic

We prove that our implementation is equivalent to a reference Coq implementation, which we will prove correct in Section V. The reference, `by_inv_ref`, is in `Ref.v`. The correctness theorem (for the Unsaturated Solinas case) is depicted in Listing 2. The assumptions are rather natural: `g_length` requires that g has the appropriate amount of limbs, `g_in_bounded` requires that each limb of g is within a word size and `g_bounds` requires that g is less than half the absolute max value representable in two's complement. We ensure that this is true when g is a number mod m, by choosing the quantity `tc_limbs` such that $m < 2^{\text{machine\_wordsize} \cdot \text{tc\_limbs}} - 2$.

The proof of theorem required us to prove several invariants of `divstep`. Fiat-Crypto provided sufficient machinery to make this fairly simple; see e.g. `divstep_iter_bounds` in `Divstep.v`. The only potential "overflow" we have to worry about, is d, since it can in practice grow unboundedly if one iterates divstep sufficiently many times; this shows up in the lemma `divstep_iter_correct` where you have to bound d depending on how many times you iterate. The other quantities cannot overflow, since f, g are decreasing for each divstep and v, r are computed using modular arithmetic. Only in the very first iteration, when you add f and g, do you risk to overflow: this is exactly why we need to have 2 spare bits when representing the prime in two's complement (1 for this potential overflow and 1 for the sign).

Montgomery inversion. Inverting in Montgomery arithmetic needs additional care to compute the inverse wrt. Montgomery multiplication, not regular modular multiplication. This amounts to computing the regular inverse multiplied by R^2. This works since,

$$\begin{aligned}(aR \bmod M)((aR)^{-1} \cdot R^2 \bmod M)R^{-1} \bmod M \\ = 1 \cdot R \bmod M\end{aligned}$$

which indeed is 1 in the Montgomery domain. Accordingly, the theorem `eval_by_inv_jump` contains this factor in the Montgomery case.

E. *Implementing* JUMPDIVSTEPS

We provide implementations of a single jumpdivstep, i.e. the body of the for loop in Algorithm 3. The implementations are in the file `Definitions.v` and properties about them are proven in `JumpDivstep.v`. There are implementations for both the Montgomery arithmetic and unsaturated Solinas. We have

```
Definition jump_divstep_aux '(d, f, g, v, r) :=
  let '(d1,f1,g1,u1,v1,q1,r1) :=
    fold_right word_divstep
      (d,nth_default 0 f 0,nth_default 0 g 0,1,0,0,1)
      (seq 0 (machine_wordsize - 2)) in
  let f2 := word_tc_mul u1 f in
  let f3 := word_tc_mul v1 g in
  let g2 := word_tc_mul q1 f in
  let g3 := word_tc_mul r1 g in
  let f4 := tc_add word_tc_mul_limbs f2 f3 in
  let g4 := tc_add word_tc_mul_limbs g2 g3 in
  let f5 := arithmetic_shiftr f4 (machine_wordsize - 2) in
  let g5 := arithmetic_shiftr g4 (machine_wordsize - 2) in
  let f6 := firstn tc_limbs f5 in
  let g6 := firstn tc_limbs g5 in
  let u2 := twosc_word_mod_m u1 in
  let v02 := twosc_word_mod_m v1 in
  let q2 := twosc_word_mod_m q1 in
  let r02 := twosc_word_mod_m r1 in
  let v2 := mulmod u2 v in
  let v3 := mulmod v02 r in
  let r2 := mulmod q2 v in
  let r3 := mulmod r02 r in
  let v4 := addmod v2 v3 in
  let r4 := addmod r2 r3 in
(d1,f6, g6, v4, r4).
```

Listing 4: Implementation of jumpdivstep using Montgomery arithmetic.

included the implementation in Montgomery arithmetic here in Listing 4.

For JumpDivsteps we need a couple of additional methods. The idea of JumpDivsteps is computing Divsteps (line 4) on word-sized integers (we use $k = \mathrm{mw} - 2$ 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 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 `twosc_word_mod_m` does (the corresponding function for Solinas is `word_to_solina`), by computing the negation and then choosing based on sign (recall that it has to compute the negation always otherwise a branch is introduced).

Proving `jump_divstep` correct is similar to proving `by_inv`, in that we have to prove invariants about (word-sized) divsteps. As we did for the other algorithms, we prove it correct wrt. a reference, namely the function

```
Definition jump_divstep_vr
  (n : nat) (mw m : Z) '(d, f, g, v, r) :=
  let '(d1, f1, g1, u1, v1, q1, r1) :=
    iter n divstep_uvqr
      (d, f mod 2 ^ mw, g mod 2 ^ mw, 1, 0, 0, 1) in
  let f1' := (u1 * f + v1 * g) / 2 ^ n in
  let g1' := (q1 * f + r1 * g) / 2 ^ n in
  let v1' := (u1 * v + v1 * r) mod m in
  let r1' := (q1 * v + r1 * r) mod m in
  (d1, f1', g1', v1', r1').
```

Here, `iter` is just an iterator applying the second argument to itself `n` times, initialized with the third argument to `iter`. The function `divstep_uvqr` is similar to `divstep_vr_mod` from earlier, but it also computes the q and r quantities; these are needed in JumpDivsteps to update the values f, g, v and r.

```
Theorem eval_by_inv_jump g
  (g_length : length g = tc_limbs)
  (g_in_bounded : in_bounded g)
  (g_bounds :
    - 2 ^ (mw * tc_limbs - 2) <
      tc_eval g < 2 ^ (mw * tc_limbs - 2)) :
  eval (by_inv_jump g) mod m = by_inv_jump_ref (tc_eval g).
```

Listing 5: Correctness of `by_inv_jump`

The correctness of `jump_divstep` states that

```
Theorem jump_divstep_correct d f g v r (...) :
let '(d1, f1,g1,v1,r1) :=
  jump_divstep_aux (d, f, g, v, r) in
(tc d1, tc_eval f1, tc_eval g1,
       eval v1 mod m, eval r1 mod m)
  = jump_divstep_vr (mw - 2) mw m
     (tc d, tc_eval f, tc_eval g,
          eval v mod m, eval r mod m).
```

Note that the argument `mw - 2` corresponds to k in Algorithm 3, i.e. how many times we iterate divstep on word-sized integers, and the argument `mw` corresponds to where we truncate (truncating at `mw` is easy: simply take the first limb). This correctness theorem is for unsaturated Solinas arithmetic. We have omitted the preconditions to this theorem here for clarity, but they are all natural (and necessary). E.g., d has to have a distance of `mw - 2` to the lowest and highest value representable in two's complement (otherwise iterating divsteps `mw - 2` times might overflow).

a) Lazy Montgomery reduction: When we translate words to multi-limb numbers in the Montgomery arithmetic setting, we ought to multiply with R, such that we get the representations in the Montgomery domain; we can however just propagate these factors through the execution and include them in the final recomputed constant. Note that this means that these factors show up in the correctness theorem in the Montgomery setting (see the `WordByWordMontgomery` section in `JumpDivstep.v`).

F. *Implementing* BY-inversion *using* JumpDivsteps

We provide implementations of Algorithm 2 using JumpDivsteps. The implementations are in the file `Definitions.v` and properties about the implementations are proven in `BYInvJump.v`. There are implementations for Montgomery arithmetic and Unsaturated Solinas. We have included the implementation using Montgomery arithmetic here in Listing 6.

The implementation is very close to BY-inversion using Divsteps – the only changes are the number of iterations and (consequently) the precomputed value to multiply at the end.

To prove correctness of `by_inv_jump` as specified in Listing 5 we need to prove that `jump_divstep` preserves several invariants. Here we utilize that we have the reference Coq implementation and instead prove the invariants about this function and then transport them from the Coq function defined over `Z` to the one defined over `list Z` (`jump_divstep`).

We do this e.g. in `jump_divstep_invariants2` in `JumpDivstep.v` and in the Montgomery version of

```
Definition by_inv_jump g :=
  let bits := (Z.log2 m) + 1 in
  let msat := partition m in
  let jump_its := jump_iterations bits in
  let pc := partition jumpdivstep_precomp in
  let '(_, fm, _, vm, _) :=
    fold_left (fun data i => jump_divstep_aux data)
              (seq 0 (Z.to_nat jump_its))
              (1,msat,g,zero,one) in
  let sign := tc_sign_bit fm in
  let inv := mulmod pc vm in
  let inv := select sign inv (oppmod inv) in
  inv.
```

Listing 6: Implementation of BY-INVERSION using JUMPDIVSTEPS and Montgomery arithmetic

`jump_divstep_iter_correct` in `JumpDvstep`. In the first one we use `jump_divstep_vr_invariants`, proven in `Ref.v`, which asserts how the bounds of the outputs of an iterated jumpdivstep depend on the bounds of the inputs. In the second one, we use that a multiplication on inputs propagates through and corresponds to a multiplication on the outputs (used to propagate R-factors through the computation). This is lemma `nat_iter_jump_divstep_vr_mul` proven at `Ref.v`.

Differences from [20]. In Section 12 of [20], the authors compute the matrix product in Algorithm 3 by recursively dividing it into halves, resulting in a total of $n-1$ matrix multiplications. This allows them to keep the precision of the entries as low as possible.

We compute the product 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, by 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).

G. Generating Fiat-Crypto code

Thus far we have not explained how to turn our Fiat-Crypto implementation into efficient low-level code, which is not so straightforward. Even though the implementations `by_inv` (Listing 3) and `by_inv_jump` (Listing 6) are reifiable into Fiat-Crypto's internal language, they are both too large for this to be feasible. The reason is that (1) the folds used are unrolled by Fiat-Crypto, and there is currently no way to get Fiat-Crypto to generate a proper loop, and (2) each function is fully inlined, since Fiat-Crypto does support function calls. As a result, the generated code would be many thousands of lines long and the code generation would slow down prohibitively.

It is unclear if these limitations will be fixed in Fiat-Crypto, in particular supporting function calls would require a richer internal language. On the other hand, these are also not functions which Fiat-Crypto claims to be able to generate, so we are probably pushing the limit for how large programs should be synthesized. One way to fix this, would be to export Fiat-Crypto implementations along with their proofs of correctness to a richer language. Work in this direction has started using bedrock2 [25].

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

  WORD precomp[LIMBS], h[LIMBS];
  WORD f1[SAT_LIMBS], f[SAT_LIMBS], g1[SAT_LIMBS];
  WORD v1[LIMBS], v[LIMBS];
  WORD r1[LIMBS], r[LIMBS];
  WORD d, d1, its;
  uint8_t s;

  MSAT(f);
  ITERATIONS(&its);
  PRECOMP(precomp);

  for (int j = 0; j < LIMBS; j++) {
    v[j] = 0;
    r[j] = 0;
  }
  r[0] = 1;
  d = 1;

  for (int i = 0; i < its - (its % 2); i += 2) {
    DIVSTEP(&d1, f1, g1, v1, r1, d, f, g, v, r);
    DIVSTEP(&d, f, g, v, r, d1, f1, g1, v1, r1);
  }
  if (its % 2) {
    DIVSTEP(&d1, f1, g1, v1, r1, d, f, g, v, r);
    for (int k = 0; k < LIMBS; k++)
      v[k] = v1[k];
    for (int k = 0; k < SAT_LIMBS; k++)
      f[k] = f1[k];
  }

  SIGN(&s, f);
  MUL(out, v, precomp);

  OPP(h, out);
  SZNZ(out, s, out, h);

  return;
}
```

Listing 7: handwritten reassembly of the Fiat-Crypto implementation of BY-INVERSION in Listing 3

We manage to generate code by splitting our implementation into two parts: The body of fold (i.e., a single divstep and a single jumpdivstep respectively) and the functions outside the loop. For the jumpdivstep version, we unfortunately also had to split the body of the loop into three parts to make reification and code generation succeed.As a result, one has to reassemble the code manually in C; as depicted in Listing 7 and Listing 8. When doing this, one should be very careful to replicate the structure of Fiat-Crypto implementation; as depicted in Listing 3 and Listing 6. Then the correctness theorem about the assembled program should still hold, though you cannot prove this formally. That this has been done correctly should be manually verified.

H. Implementing BY-INVERSION *using half-delta* JUMPDIVSTEPS

We also provide an implementation of a faster variant of BY-INVERSION proposed by Wuille *et al.* [21]. It is

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

  WORD precomp[LIMBS], h[LIMBS];
  WORD f1[SAT_LIMBS], f[SAT_LIMBS], g1[SAT_LIMBS];
  WORD v1[LIMBS], v[LIMBS];
  WORD r1[LIMBS], r[LIMBS];
  WORD d, d1, un, vn, qn, rn, its;
  uint8_t s;

  MSAT(f);
  ITERATIONS(&its);
  PRECOMP(precomp);

  for (int j = 0; j < LIMBS; j++) {
    v[j] = 0;
    r[j] = 0;
  }
  r[0] = 1;
  d = 1;

  for (int i = 0; i < its - (its % 2); i += 2) {
    FN_INNER_LOOP(&d1, &un, &vn, &qn, &rn, d, f, g);
    UPDATE_FG(f1, g1, f, g, un, vn, qn, rn);
    UPDATE_VR(v1, r1, v, r, un, vn, qn, rn);

    FN_INNER_LOOP(&d, &un, &vn, &qn, &rn, d1, f1, g1);
    UPDATE_FG(f, g, f1, g1, un, vn, qn, rn);
    UPDATE_VR(v, r, v1, r1, un, vn, qn, rn);
  }
  if (its % 2) {
    FN_INNER_LOOP(&d1, &un, &vn, &qn, &rn, d, f, g);
    UPDATE_FG(f1, g1, f, g, un, vn, qn, rn);
    UPDATE_VR(v1, r1, v, r, un, vn, qn, rn);

    for (int k = 0; k < LIMBS; k++)
      v[k] = v1[k];
    for (int k = 0; k < SAT_LIMBS; k++)
      f[k] = f1[k];
  }

  SIGN(&s, f);
  MUL(out, v, precomp);

  OPP(h, out);
  SZNZ(out, s, out, h);

  return;
}
```

Listing 8: handwritten reassembly of the Fiat-Crypto implementation of BY-INVERSION in Listing 6

the same as `by_inv_jump` but with slightly different constants, see `jump_divstep_hd` and `jump_divstep_precomp_hd` in `Definitions.v`. 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 formal correctness proof in the latest version of the repository for the result, we understand this has neither been peer-reviewed nor formalized. So, we take the extra precaution of validating the lower number of iterations. We adapted their 256-bit Coq proofs 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 [26] to OCaml native binaries for another 2-factor reduction in time. Table I reports on the time for running all proofs. At the moment, there is no connection between our implementation in Fiat-Crypto and this formalized proof. We would merely get an equivalence between the Fiat-Crypto implementation and a reference Coq implementation.

We do not prove properties about the half-delta BY-INVERSION implementation, though one could easily adapt the proofs of correctness of the other implementations to this one. The reason we did not do this, is that in the end we would not get an end-to-end proof, since the method we use to formalize the reference implementation in Section V is not capable of proving this lower bound of iterations, see [20] section 8.

TABLE I
TIME TAKEN TO RUN COMPUTATIONAL PROOFS TO VALIDATE THE NUMBER OF ITERATIONS FOR VARIOUS PRIME MODULI SIZES IN THE HALF-DELTA OPTIMIZATION USING DIFFERENT STRATEGIES.

Size (bits)	Iterations	Coq-`native`	Coq-`ExtOCaml`
256	590	32.1 hours	14.7 hours
381	878	213.0 hours	100.5 hours
448	1033	634.3 hours	281.1 hours
521	1201	1226.7 hours	557.6 hours
575	1325	2671.5 hours	906.7 hours

I. *Experimental results*

We have generated and benchmarked field inversion in C for primes commonly used in both ECC and PBC settings of curve-based cryptography. For ECC, 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 respectively the 128-, 224- and 256-bit security levels. For PBC, we took the base fields for Barreto-Naehrig (BN) [7] and Barreto-Lynn-Scott (BLS) curves [27] at three different security levels. These are the 254-bit prime used in the now legacy 110-bit secure BN curves [10], [28], the 381-bit prime for BLS curves with embedding degree 12 undergoing standardization at 128-bit security [28], and the 575-bit prime for BLS curves with embedding degree 48 proposed for 256-bit security [29].

The generated code was integrated in the RELIC toolkit [30], a cryptographic library containing several state-of-the-art implementations of pairings. RELIC uses a combination of handwritten assembly (ASM) with higher-level C-code and has set speed records for several of the PBC parameters. Integrating the code within RELIC allowed convenient comparison between the efficiency of our approach and other field inversion algorithms already implemented in the library. We benchmarked the implementations on an Intel Skylake Core i7-6700K CPU running at 4.00GHz, using GCC version 12.1 and `clang` from LLVM 13. Numbers were obtained by computing the average of 10^4 consecutive executions measured using the cycle counter. TurboBoost and HyperThreading were disabled for benchmarking stability.

We present our results in Table II and Table III. In both tables, the first part has baseline implementations from the GMP 6.2.1 library [31] used for reference. With the exception of *Variable-time GMP*, all operations are implemented in constant-time. These timings set a lower bound (aggressively optimized variable-time code) and upper bound (generic constant-time approach) that illustrate how challenging implementing efficient field inversion in constant-time can be 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 [32], [33] (FLT+ASM) and benchmarked the same addition chains over field arithmetic generated by Fiat-Crypto (FLT+Fiat). For PBC, we built and benchmarked RELIC using both the existing ASM backends and field arithmetic code generated by Fiat-Crypto. Since the same number of multiplications are executed in FLT, the timings illustrate the penalty of going from handwritten ASM to Fiat-Crypto for the different parameters: an approximate slowdown of 1.2–3.0 when compiling using either `clang` or GCC.

The remaining rows in the tables show the performance of our various implementations of BY-INVERSION. The most interesting entries performance-wise are jumpdivstep and hdjumpdivstep, respectively the jumpdivstep implementation that we generate automatically from Fiat-Crypto; and the half-deta variant proposed later. These are also benchmarked in RELIC in the PBC case using the provided ASM backends.

We compare performance against FLT due to its generality, and acknowledge that performance speedups are due in part to choice of algorithms. For ECC, the hdjumpdivstep implementations outperform the FLT implementations in the two largest primes, showing that FLT approaches do not scale well for larger parameters, with speedups over unverified assembly (FLT+ASM) of 14% for Curve448 and 13% for P521. For PBC, the speedup over FLT+ASM is visible in all fields and grows to the range between 39%–49%. We also outperform FLT over a verified multiplier (FLT+Fiat) by up to a 2-factor in all cases, except Curve25519. When comparing the fastest verified implementations of BY-INVERSION with our implementation within RELIC using its unverified ASM backends (RELIC+ASM), the performance difference ranges from 2 to 5.3, a tolerable trade-off considering the correctness guarantees provided by the Fiat-Crypto version.

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 [6], [34]. The Bernstein-Yang paper reports 8,778 Skylake cycles for inversion, later improved to 3,900 cycles[8]. An alternative approach by Thomas Pornin [35] 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 handwritten assembly optimizations including vector instructions. We benchmarked inversion from the C+ASM verified implementation of Curve25519 in EverCrypt [36] at 12,728 cycles in the same machine, which gives us a small 15% difference in latency. For reference, Fiat-Crypto is 21% slower than Evercrypt according to the original benchmarks.

```
Definition iterations b :=
  if b <? 46 then (49 * b + 80) / 17
    else (49 * b + 57) / 17.
Definition jump_iterations b mw :=
  ((iterations b) / (mw - 2)) + 1.

Definition by_inv_ref (f g : Z) :=
  let bits := Z.log2 f + 1 in
  let i := iterations bits in
  let k := (f + 1) / 2 in
  let pc := (k ^ i) mod f in
  let '(_, fm, _, vm, _) :=
    iter its (divstep_vr_mod f) (1, f, g, 0, 1)  in
  let sign := if fm <? 0 then (-1) else 1 in
  sign * pc * vm mod f.

Definition by_inv_jump_ref mw f g :=
  let bits := (log2 f) + 1 in
  let its := jump_iterations bits mw in
  let total_iterations := its * (mw - 2) in
  let k := (f + 1) / 2 in
  let pc := (k ^ total_iterations) mod f in
  let '(_, fm, _, vm, _) := iter its (jump_divstep n mw f)
  ↪  (1, f, g, 0, 1)  in
  let sign := if fm <? 0 then (-1) else 1 in
  sign * pc * vm mod f.
```

Listing 9: Implementation of BY-INVERSION in Coq

```
Theorem by_inv_spec f g
  (f_bound : (21 < log2 f))
  (g_bound : 0 < g <= f)
  (fg_rel_prime : gcd f g = 1)
  (f_odd : odd f = true) :
by_inv_ref f g * g mod f = 1.

Theorem by_inv_jump_spec mw f g
  (f_bound : (21 < log2 f))
  (g_bound : 0 < g <= f)
  (mw_bound : 2 < mw)
  (fg_rel_prime : gcd f g = 1)
  (f_odd : odd f = true) :
by_inv_jump_ref mw f g * g mod f = 1.
```

Listing 10: Correctness of BY-INVERSION in Coq

V. Formalization of Bernstein-Yang inversion

This section presents the formalization of Theorem 1, which proves the correctness of the BY-INVERSION algorithm (Algorithm 2). The development is available at https://github.com/bshvass/by-inversion and paths in this section will be relative to `by-inversion/src/`.

The two Coq versions of the algorithm are in Listing 9 (using DIVSTEPS and JUMPDIVSTEPS respectively) and the main theorems of the development, which state that the Coq algorithms are correct, are in Listing 10.

A. *p-adic valuations*

Since some theory of p-adic valuations was required for the proof, we developed a small library for the basics of this theory. This is in the file `PadicVal.v`. We implemented the p-adic valuation by simply counting the number of times a number is divisible by p and proved the following specification

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

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

TABLE II

Benchmarks of different approaches for field inversion over ECC fields. Numbers in bold are the fastest for group of implementations in this work or related work among the different compilers for a certain choice of prime.

	Verified	Curve25519		Curve448		NIST-P521	
		`clang`	`gcc`	`clang`	`gcc`	`clang`	`gcc`
Variable-time GMP	No	3,098	3,314	4,724	5,799	6,814	7,128
Constant-time GMP	No	75,895	76,300	186,637	186,186	257,935	270,085
FLT+ASM	No	–	**9,301**	***41,400**	–	–	**53,828**
FLT+Fiat	Partially	13,638	11,778	49,867	46,103	78,565	55,139
This work+Fiat (divstep)	Yes	69,942	72,583	189,542	248,685	230,221	308,478
This work+Fiat (jumpdivstep)	Yes	17,797	20,394	43,529	53,186	57,894	67,928
This work+Fiat (hdjumpdivstep)	Yes	**14,652**	17,166	**35,549**	43,401	**46,747**	54,708

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

TABLE III

Benchmarks of different approaches for field inversion over PBC fields. Numbers in bold are the fastest for group of implementations in this work or related work among the different compilers for a certain choice of prime.

	Verified	BN-254		BLS12-381		BLS48-575	
		`clang`	`gcc`	`clang`	`gcc`	`clang`	`gcc`
Variable-time GMP	No	3,291	3,270	4,724	4,716	7,495	7,504
Constant-time GMP	No	75,639	76,168	146,157	146,083	270,631	271,168
FLT+ASM	No	**31,452**	31,492	104,513	**103,361**	288,719	**288,109**
FLT+Fiat	Partially	57,748	75,283	182,825	262,919	577,626	856,437
RELIC+ASM (divstep)	No	87,456	87,584	167,724	164,459	335,864	337,641
RELIC+ASM (jumpdivstep)	No	14,382	14,383	23,820	23,810	43,941	43,989
RELIC+ASM (hdjumpdivstep)	No	**9,777**	9,873	16,377	**16,183**	31,963	**27,911**
This work+Fiat (divstep)	Yes	80,018	120,486	172,497	296,065	390,773	671,370
This work+Fiat (jumpdivstep)	Yes	23,406	30,572	62,555	75,412	180,023	220,852
This work+Fiat (hdjumpdivstep)	Yes	**19,733**	25,836	**52,628**	63,237	**147,402**	180,530

i.e. that `pval p a` is the maximal power of p which divides a. We also prove uniqueness such that the full specification becomes

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

We also define split p, which divides a number by the maximal power of p which divides it evenly. We prove the specification

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

B. The gcd algorithm

The formalization of the gcd algorithm using divsteps (described in Appendix E in [20]) is in `AppendixE.v`. We prove two main results:

1) The existence and specification of $q(f,g)$ (Theorem E.1 in [20])
2) The correctness of the gcd algorithm assuming termination (Theorem E.3 in [20])

We implemented a function computing $q(f,g)$ instead of proving its existence abstractly. This allows us to use it in the recursive definition of R_i (see Theorem E.2 in [20]). One minor issue here is that to construct $q(f,g)$ one needs to compute the inverse of $\text{split}_2\, g$ 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 in $\mathbb{Z}$), but when constructing $q(f,g)$ one only needs the inverse to a finite precision. This is done by computing the inverse of $\text{split}_2\, g$ modulo 2^i. We recall that Coq's default logic and type theory are constructive. This has the advantage that all definable functions are actually computer programs, and allows us to carry out proofs by computation. It is possible to consistently add the axiom of excluded middle, but as a result the computation may get stuck. We have decided to use such non-computable real numbers. This allows us to carry out all operations in the same type (`R`), as opposed to silently coercing (embedding) rational numbers into the real numbers. For example, the expression $f\, \text{div}_2\, g$ is rational, but we give it type `R`:

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

where `IZR` is the embedding from the `Z` to `R` (see also Theorem E.1 in [20]).

A constructive solution could have been to use a library of algebraic, or constructive, real numbers.

C. Complexity analysis

The formalization of the termination proof of the gcd algorithm is in `AppendixF.v`. This part of the proof is the content of Appendix F in [20]. Here, their proof becomes more complicated and one theorem from [20] depends on the termination of a SAGE program (see theorem F.22 and figure F.23 in [20]).

The mathematical community is ambivalent about such 'computer proofs'; see, for example, the discussions around the 4-color theorem [37] and Kepler's conjecture [38]. A popular solution is to carry out the computation inside a proof assistant. Although Coq has become much faster in recent years, e.g. due to addition of native compilation [39], this computation is still beyond the scope of what can be done in Coq in reasonable time. Instead, we use Coq's extraction mechanism [26] to translate the Coq program to a related program in OCaml. This slightly extends the *trusted computing base*. That is, to trust formalization of the proof is correct, one also has to trust the unverified extraction mechanism. However, possible bugs in extraction are likely to be orthogonal to possible issues in [20].

The operator norm. To prove termination of the gcd algorithm, we need to prove that the operator norm of products of transition matrices is bounded by an exponentially decreasing sequence of numbers. In particular, we have to introduce the operator norm of matrices; we only define it for 2 by 2 matrices since this suffices for the proof. To this end, we use the following formula

```
Definition mat_norm (m : mat) :=
  let '(m11, m12, m21, m22) := m in
  let a := (m11 ^ 2 + m12 ^ 2)%R in
  let b := (m11 * m21 + m12 * m22)%R in
  let c := (m11 * m21 + m12 * m22)%R in
  let d := (m21 ^ 2 + m22 ^ 2)%R in
  sqrt ((a + d + sqrt ((a - d) ^ 2 + 4 * b ^ 2)) / 2).
```

To prove that this definition enjoys properties such as $|Mv|_2 \leq |M||v|_2$ and $|MN|_2 \leq |M|_2|N|_2$, we prove and use the spectral theorem for 2 by 2 real matrices. This is the content of `Spectral.v`. See also theorem F.11 in [20] (note that by taking this formula as our definition, we do not need to prove theorem F.11).

The file `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 `Hierarchy/`. This library is in the style of math-classes [40], although in the interest of simplicity we do not depend on it. For similar reasons, we did not reuse [41]. 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.

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

```
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 [20], which is formalized in `AppendixF.v`. The formalized proof is a little laborious, since we did not find any simple tactics for manipulating expressions involving square roots (`sqrt`). Our general strategy is to reduce an expression to an expression without square roots by isolating and squaring appropriately. These two methods do not, however, suffice in general (consider e.g. $\sqrt{5} \leq 1 + \sqrt{2} + \sqrt{3}$). A general solution would be to construct the ring of integers extended with square roots as a subring of the real closed field extending the rationals and use advanced decision procedures there [42].

The bounding sequence. Next, we define the number sequence α_n which will bound the operator norms. Using this we prove three 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 [20]).
- That all matrices in this particular subset are bounded by α_n (Theorem F.22 in [20]).
- That the gcd algorithm terminates (Theorem F.26 in [20]).

Now we use Coq's extraction mechanism to prove Theorem F.22, as discussed at the beginning of this section. The development of this computational proof is the contents of `Comp1`. The file `Comp1/Mem.v` contains the program to be extracted (`depth_first_verify`) which utilizes memoization to achieve performance. Memoization complicated the formalization quite a bit, and we split the proof as follows:

1) Proving that the memoized program terminates (using extraction).
2) Proving the memoized program equivalent to a non-memoized program (this is in `Comp1/Mem.v`).
3) Proving that if the non-memoized program terminates, then theorem F.22 follows (this is the content of `Comp1/NoMemNew.v`).

Running the extracted program terminates[9] and outputs the same result as reported in [20]. The axiom which we add is `comp1_theorem` in `Comp1/Mem.v`.

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 `AppendixE.v`).

We developed a small "big operator" library [43] to reason about the big multiplications ($\prod$) and big additions ($\sum$), as required in theorem F.21. Like our theory of matrices, it is built on top of `Hierarchy.v`. The same reason for developing our *own* matrix library applies here, we wanted to have a greater degree of control over the implementation.

D. 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 `AppendixG.v`. We have also included an

[9] In 332 minutes using OCaml 4.11.1 on a Intel Coffee Lake Core i7-9750H.

implementation of the inversion algorithm, Algorithm 2, in `BYInv.v` with an accompanying correctness proof. Note that we prove a slightly specialized version of Theorem 1: We require that the maximum bitwidth of f and g to be at least 21. This suffices for all cryptographically interesting primes and, in particular, the primes which we have benchmarked in Section IV-I.

A formal proof of correctness for inputs of bounded size exists [21]. Their approach is different from ours, in that they generate bounds and proofs of correctness from bounds on the inputs (which the caller provides). They achieve this by using a new proof strategy using convex hulls of all possible branches in a sequence of divsteps. Their proofs, like ours, use very heavy computation based on a program written in Coq. Unlike ours, the time it takes for their algorithm to finish is (barely) feasible within Coq.

E. Assumptions and trusted computing base (TCB)

Fiat-Crypto is axiom free but the printer from the intermediate language to C is not verified, and thus it has to be trusted. The formalization of the proof of BY-INVERSION depends on a few axioms; these are printed at end of building the development. The first four are standard axioms from the classical reals in the standard library, which are needed because Coq's logic is constructive: `sig_not_dec` (axiom of limited omniscience), `sig_forall_dec`, `functional_extensionality_dep` (functional extensionality) and `classic` (law of excluded middle). The last axiom `comp1_theorem` is the assertion that the computation of the term `depth_first_verify` terminates and yields `Some` number (as described in Section V-C). We verify this last axiom using extraction to OCaml, adding the extraction mechanism of Coq to the TCB. This is a small, but standard, extension of the TCB, similar to Coq's **`native_compute`** [39], which also uses the ocaml compiler after a translation of a Coq term to an OCaml program. The precise term is

```
comp1_theorem : depth_first_verify = Some 3787975117
```

VI. Connecting the formalization of BY-inversion and Fiat-Crypto

Having proven both Listing 2 and Listing 10 we can combine them and prove that Fiat-Crypto does in fact compute the inverse (wrt. the modular operations in Fiat-Crypto).

However, since the theorems are proven under different developments, we have to copy the correctness theorem Listing 10 to the Fiat-Crypto development, and include it as an assumption in our assertions. We hesitate to merge the two libraries, as it would quite a large dependency to the already substantial Fiat-Crypto library – however the propositions are identical as can be inspected (compare `Ref.v` in the Fiat-Crypto repository to `BYInv.v` in the by-inversion repository).

Converting representations. There is a technical caveat when comparing the input of `by_inv` to its output: The function `by_inv` assumes that its input g is represented as a multi-limb number in two's complement, and it ensures that its output is in either Montgomery or unsaturated Solinas representation (depending on the implementation). Usually, you will want both input and output to be in the appropriate representation for modular arithmetic (Montgomery/Solinas). Thus, to use the function, we have to convert between representations first. We will write `to_tc` for this conversion (to two's complement).

Word by word Montgomery. For multi-limb numbers in Montgomery form this function is simply zero-extending by a single limb if necessary, i.e.

```
Definition to_tc := extend_to_length mont_limbs tc_limbs.
```

where `extend_to_length` is provided by Fiat-Crypto, `mont_limbs` is the amount of limbs in Montgomery representation and `tc_limbs` is amount of limbs needed for representing in two's complement. Recall that we need 2 more bits to represent m in two's complement, so this will just be the identity, unless m's is within 2 bits of a multiple of the machine word size. Using this, we can prove the theorem

```
Lemma by_inv_correct g
  (g_length : length g = mont_limbs)
  (g_nonzero : eval g <> 0)
  (g_valid : valid g)
  (spec : by_inv_spec m (eval g)) :
eval (mulmod (by_inv (to_tc g)) g) mod m
  = 2 ^ (machine_wordsize * mont_limbs) mod m.
```

by applying the general lemmas about `mulmod` and the correctness theorem of `by_inv` (Listing 2). The `valid` predicate on g asserts that the evaluation of g is less than m and that it is in the unique saturated representation. This is required for all Montgomery operations and is the responsibility of the caller. Note that g is not invertible if g is zero and that `2 ^ (machine_wordsize * mont_limbs) mod m` *is* indeed the identity in the Montgomery domain.

We prove an identical theorem about `by_inv_jump` in `BYInvJump.v`.

Unsaturated Solinas. For the unsaturated representation, we have to be a bit more careful, since the representation is not unique. If we just naively convert to two's complement, we might get an "overflow" error, where a positive integer incorrectly becomes interpreted as negative. We therefore utilize the `freeze` function provided by Fiat-Crypto to get a canonical representation, which can safely be converted to two's complement. The definition becomes

```
Definition to_sat mw num den n tc_limbs m g :=
  let g := freeze (weight num den) n (ones mw) m g in
  convert_bases num den mw 1 n tc_limbs g.
```

Here $\mathrm{num}/\mathrm{den}$ is the base of the unsaturated representation and n is the amount of limbs, such that `convert_bases` converts from the unsaturated representation to two's complement, which has base $\mathrm{mw}/1$ and limbwidth `tc_limbs`, as expected. We are then able to prove the theorem

```
Lemma by_inv_correct g
  (g_length : length g = n)
  (g_nonzero : eval g mod m <> 0)
  (g_bounds : 0 < eval g < 2 * m)
  (spec : by_inv_spec m (eval g mod m)) :
eval (carry_mulmod (by_inv (to_tc g)) g) mod m = 1.
```

again by combining previous results. The bound requirements on g are needed for the correctness of `freeze`. They are the responsibility of the caller, but can always be met by calling the unsaturated Solinas function `carrymod`, which ensures that its output is within these bounds.

Additional constraints on m. Note that we need some assumptions about the prime m in addition to the ones already required by Fiat-Crypto. As mentioned in Section IV, we need that `m < 2 ^ (machine_wordsize * tc_limbs - 2)` to ensure that we can interpret m correctly in two's complement. Secondly, we need that $21 < \log_2 m$, which is true for all cryptographic use cases. Finally we need to prove that the amount of times we iterate in the implementations of BY-INVERSION do not result in overflows. The concrete bounds are in `BYInv.v` and `BYInvJump.v`, named `iterations_bounds`. These constraints are all checked before code generation (see `check_args` in `UnsaturatedSolinas.v` and `WordByWordMontgomery.v`, in `src/PushButtonSynthesis`).

VII. Related work

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

EverCrypt [36] 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* [22], which is a collection of cryptographic protocols implemented, specified and verified in a subset of 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 [44]. 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 [45], which provides a higher level assembly language and a verified compiler to Intel assembly. Jasmin has been used to verify a high-performance SHA3 implementation [46], among many other primitives [47].

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 depend on an unverified reduction to unverified SMT-solvers. Moreover, Easycrypt does not include evaluation of functions. We are not aware that F* would be able to complete the computation that Coq cannot.

VIII. 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. 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-Crypto is geared for this, but if it is, then the implementation should be no more difficult than what is presented in Section IV.

Going even further, one could extend Fiat-Crypto to generate elliptic curve arithmetic directly. However, Fiat-Crypto's low-level language does not include function calls and these are necessary to implement elliptic curve operations. A solution could be to the use bedrock2, a low-level language embedded in Coq with links to the Fiat-Crypto library. This has been pursued in [25].

After generating elliptic curve arithmetic, we can go yet another step up the abstraction layer and implement bilinear 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 [48] and in the Fiat-Crypto library, but no formalized implementation of a pairing has been publicly developed.

IX. Acknowledgements

We would like to thank Yifan Zheng for finding typos in Algorithm 1, Pieter Wuille and Tim Ruffing for interesting discussions about their related work and the anonymous reviewers for the suggested improvements. Part of this research was supported by the Concordium Foundation.

References

[1] A. C. Aldaya, A. C. Sarmiento, and S. Sánchez-Solano, "SPA vulnerabilities of the binary extended euclidean algorithm," *J. Cryptogr. Eng.*, vol. 7, no. 4, pp. 273–285, 2017.

[2] A. C. Aldaya, C. P. García, L. M. A. Tapia, and B. B. Brumley, "Cache-timing attacks on RSA key generation," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2019, no. 4, pp. 213–242, 2019.

[3] A. C. Aldaya, C. P. García, and B. B. Brumley, "From A to Z: projective coordinates leakage in the wild," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2020, no. 3, pp. 428–453, 2020.

[4] D. Belyavsky, B. B. Brumley, J. Chi-Domínguez, L. Rivera-Zamarripa, and I. Ustinov, "Set it and forget it! turnkey ECC for instant integration," in *ACSAC*. ACM, 2020, pp. 760–771.

[5] J. W. Bos, "Constant time modular inversion," *J. Cryptogr. Eng.*, vol. 4, no. 4, pp. 275–281, 2014.

[6] D. J. Bernstein, "Curve25519: New Diffie-Hellman Speed Records," in *PKC*, ser. LNCS, vol. 3958. Springer, 2006, pp. 207–228.

[7] P. S. L. M. Barreto and M. Naehrig, "Pairing-Friendly Elliptic Curves of Prime Order," in *SAC*, ser. LNCS, vol. 3897. Springer, 2005, pp. 319–331.

[8] R. Barbulescu and S. Duquesne, "Updating key size estimations for pairings," *J. Cryptol.*, vol. 32, no. 4, pp. 1298–1336, 2019.

[9] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of computation*, vol. 44, no. 170, pp. 519–521, 1985.

[10] D. F. Aranha, K. Karabina, P. Longa, C. H. Gebotys, and J. López, "Faster Explicit Formulas for Computing Pairings over Ordinary Curves," in *EUROCRYPT*, ser. LNCS, vol. 6632. Springer, 2011, pp. 48–68.

[11] D. Boneh, B. Lynn, and H. Shacham, "Short signatures from the weil pairing," *J. Cryptol.*, vol. 17, no. 4, pp. 297–319, 2004. [Online]. Available: https://doi.org/10.1007/s00145-004-0314-9

[12] E. Ben-Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza, "Zerocash: Decentralized Anonymous Payments from Bitcoin," in *S&P'14*. IEEE Computer Society, 2014, pp. 459–474.

[13] B. B. Brumley, M. Barbosa, D. Page, and F. Vercauteren, "Practical Realisation and Elimination of an ECC-Related Software Bug Attack," in *CT-RSA*, ser. LNCS, vol. 7178. Springer, 2012, pp. 171–186.

[14] A. Erbsen, J. Philipoom, J. Gross, R. Sloan, and A. Chlipala, "Simple high-level code for cryptographic arithmetic - with proofs, without compromises," in *S& P*. IEEE, 2019, pp. 1202–1219.

[15] J. Blessing, M. A. Specter, and D. J. Weitzner, "You really shouldn't roll your own crypto: An empirical study of vulnerabilities in cryptographic libraries," 2021.

[16] D. F. Aranha, L. Fuentes-Castañeda, E. Knapp, A. Menezes, and F. Rodríguez-Henríquez, "Implementing Pairings at the 192-Bit Security Level," in *Pairing*, ser. LNCS, vol. 7708. Springer, 2012, pp. 177–195.

[17] D. F. Aranha, P. S. L. M. Barreto, P. Longa, and J. E. Ricardini, "The Realm of the Pairings," in *Selected Areas in Cryptography*, ser. LNCS, vol. 8282. Springer, 2013, pp. 3–25.

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

[19] D. F. Aranha, E. Pagnin, and F. Rodríguez-Henríquez, "LOVE a pairing," in *LATINCRYPT*, ser. LNCS, vol. 12912. Springer, 2021, pp. 320–340.

[20] D. J. Bernstein and B. Yang, "Fast constant-time gcd computation and modular inversion," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2019, no. 3, pp. 340–398, 2019.

[21] P. Wuille, G. Maxwell, and R. O'Connor, "Bounds on divsteps iterations in safegcd," https://github.com/sipa/safegcd-bounds, 2021.

[22] J. K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche, "HACL*: A Verified Modern Cryptographic Library," in *CCS*. ACM, 2017, pp. 1789–1806.

[23] T. C. D. Team, "The Coq proof assistant, version 8.12.0," Jul. 2020. [Online]. Available: https://doi.org/10.5281/zenodo.4021912

[24] J. A. Solinas, "Generalized Mersenne numbers," CACR, Tech. Rep., 1999. [Online]. Available: https://cacr.uwaterloo.ca/techreports/1999/corr99-46.pdf

[25] R. Holdsbjerg-Larsen, M. Milo, and B. Spitters, "A verified pipeline from a specification language to optimized, safe rust," in *CoqPL 2022*, 2022. [Online]. Available: https://popl22.sigplan.org/details/CoqPL-2022-papers/5/

[26] P. Letouzey, "A new extraction for Coq," in *TYPES*, ser. LNCS, vol. 2646. Springer, 2002, pp. 200–219.

[27] P. S. L. M. Barreto, B. Lynn, and M. Scott, "Constructing elliptic curves with prescribed embedding degrees," in *SCN*, ser. LNCS, vol. 2576. Springer, 2002, pp. 257–267.

[28] A. Menezes, P. Sarkar, and S. Singh, "Challenges with assessing the impact of NFS advances on the security of pairing-based cryptography," in *Mycrypt*, ser. LNCS, vol. 10311. Springer, 2016, pp. 83–108.

[29] N. B. Mbang, D. F. Aranha, and E. 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.*, vol. 4, no. 1, pp. 45–59, 2020.

[30] 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.

[31] T. Granlund and the GMP development team, *GNU MP: The GNU Multiple Precision Arithmetic Library*, 6th ed., 2012, https://gmplib.org/.

[32] K. Nath and P. Sarkar, "Efficient arithmetic in (pseudo-)mersenne prime order fields," *Adv. Math. Commun.*, vol. 16, no. 2, pp. 303–348, 2022.

[33] A. Faz-Hernández, J. C. López-Hernández, and R. Dahab, "High-performance implementation of elliptic curve cryptography using vector instructions," *ACM Trans. Math. Softw.*, vol. 45, no. 3, pp. 25:1–25:35, 2019.

[34] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B. Yang, "High-speed high-security signatures," *J. Cryptogr. Eng.*, vol. 2, no. 2, pp. 77–89, 2012.

[35] T. Pornin, "Optimized binary gcd for modular inversion," Cryptology ePrint 2020/972, 2020.

[36] J. Protzenko, B. Parno, A. Fromherz, C. Hawblitzel, M. Polubelova, K. Bhargavan, B. Beurdouche, J. Choi, A. Delignat-Lavaud, C. Fournet, N. Kulatova, T. Ramananandro, A. Rastogi, N. Swamy, C. M. Wintersteiger, and S. Z. Béguelin, "Evercrypt: A fast, verified, cross-platform cryptographic provider," in *S& P*. IEEE, 2020, pp. 983–1002.

[37] G. Gonthier, "Formal proof–the four-color theorem," *Notices of the AMS*, vol. 55, no. 11, pp. 1382–1393, 2008.

[38] T. Hales, M. Adams, G. Bauer, T. D. Dang, J. Harrison, H. Le Truong, C. Kaliszyk, V. Magron, S. McLaughlin, T. T. Nguyen *et al.*, "A formal proof of the kepler conjecture," in *Forum of Mathematics, Pi*, vol. 5. CUP, 2017.

[39] M. Boespflug, M. Dénès, and B. Grégoire, "Full reduction at full throttle," in *CPP*, ser. LNCS, vol. 7086. Springer, 2011, pp. 362–377.

[40] B. Spitters and E. van der Weegen, "Type classes for mathematics in type theory," *MSCS, special issue on 'Interactive theorem proving and the formalization of mathematics'*, vol. 21, pp. 1–31, 2011.

[41] A. Mahboubi and E. Tassi, *Mathematical Components*. Zenodo, 2020. [Online]. Available: https://doi.org/10.5281/zenodo.4282710

[42] C. Cohen and A. Mahboubi, "Formal proofs in real algebraic geometry: from ordered fields to quantifier elimination," *Log. Methods Comput. Sci.*, vol. 8, no. 1, 2012.

[43] Y. Bertot, G. Gonthier, S. O. Biha, and I. Pasca, "Canonical big operators," in *TPHOLs*, ser. LNCS, vol. 5170. Springer, 2008, pp. 86–101.

[44] B. Bond, C. Hawblitzel, M. Kapritsos, K. R. M. Leino, J. R. Lorch, B. Parno, A. Rane, S. T. V. Setty, and L. Thompson, "Vale: Verifying high-performance cryptographic assembly code," in *USENIX Security Symposium*, 2017, pp. 917–934.

[45] J. B. Almeida, M. Barbosa, G. Barthe, A. Blot, B. Grégoire, V. Laporte, T. Oliveira, H. Pacheco, B. Schmidt, and P. Strub, "Jasmin: High-assurance and high-speed cryptography," in *CCS*. ACM, 2017, pp. 1807–1823.

[46] J. B. Almeida, C. Baritel-Ruet, M. Barbosa, G. Barthe, F. Dupressoir, B. Grégoire, V. Laporte, T. Oliveira, A. Stoughton, and P. Strub, "Machine-checked proofs for cryptographic standards: Indifferentiability of sponge and secure high-assurance implementations of SHA-3," in *CCS*. ACM, 2019, pp. 1607–1622.

[47] J. B. Almeida, M. Barbosa, G. Barthe, B. Grégoire, A. Koutsos, V. Laporte, T. Oliveira, and P. Strub, "The last mile: High-assurance and high-speed cryptographic implementations," in *IEEE S & P*. S & P, 2020, pp. 965–982.

[48] E. Bartzia and P. Strub, "A formal library for elliptic curves in the Coq proof assistant," in *ITP*, ser. LNCS, vol. 8558. Springer, 2014, pp. 77–92.