

Functional Correctness of an Optimized Modular Inversion Algorithm

Assia Mahboubi 

Nantes Université, École Centrale Nantes, CNRS, INRIA, LS2N, UMR 6004, France
Vrije Universiteit Amsterdam, The Netherlands

Guillaume Melquiond 

Inria, ENS de Lyon, CNRS, Université Lyon 1, LIP, UMR 5668, Lyon, France

Pierre-Yves Strub 

PQShield, Paris, France

Tomás Vallejos Parada 

Nantes Université, École Centrale Nantes, CNRS, INRIA, LS2N, UMR 6004, France

Abstract

This article describes the first mechanized proof of functional correctness of an algorithm due to Pornin (2020), for computing modular inverses via an optimized extended binary GCD algorithm. This algorithm is widely used in cryptography applications, due to its speed and constant-timeness. But this speed comes from the use of approximate computations during its loop iterations. In particular, the pen-and-paper proof of the fact that sufficiently many loop iterations were performed is especially intricate (and the originally published version was actually wrong), which negatively impacts the trust in the applications that rely on the algorithm. In this work, we expand the notes provided in the original description by Pornin into a complete formal proof. We discuss the challenges raised by its mechanization, which eventually relies on the collaboration of deductive program verification and interactive theorem proving through the use of the tools Rocq and Why3.

2012 ACM Subject Classification Software and its engineering → Formal software verification; Theory of computation → Logic and verification; Security and privacy → Logic and verification; Security and privacy → Mathematical foundations of cryptography; Theory of computation → Cryptographic primitives

Keywords and phrases deductive program verification, modular inversion algorithm, formal verification, Rocq, Why3

Digital Object Identifier 10.4230/LIPIcs.ITP.2026.9

Supplementary Material *Software*: <https://github.com/Tvallejos/pornin-modular-inversion-is-correct>, archived at `swb:1:dir:612d5fef54a0c488ea45fe6ee7bb1fff3b4ba2ed`

Funding This project has received funding from the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation programme (grant agreement No. 101001995).

Declaration about GenAI/LLM use Generative artificial intelligence (*e.g.*, large language models) were not used at any point for this work, neither during the development/formalization stage nor for the writing of this article, not even to improve its readability.

1 Introduction

Modular inversion is a core arithmetic operation in modern public-key cryptography. It is required in widely deployed primitives such as RSA and ECC (Elliptic Curve Cryptography). In elliptic-curve systems in particular, modular inversion appears in signature generation, scalar arithmetic, and coordinate normalization over prime fields. Although certain coordinate systems can defer or reduce the number of inversions, it remains an indispensable operation in



© Assia Mahboubi, Guillaume Melquiond, Pierre-Yves Strub, and Tomás Vallejos Parada;
licensed under Creative Commons License CC-BY 4.0

17th International Conference on Interactive Theorem Proving (ITP 2026).

Editors: Ekaterina Komendantskaya and Tobias Nipkow; Article No. 9; pp. 9:1–9:19

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

performance-critical code paths. Because modular inversion is substantially more expensive than addition or multiplication in large prime fields, its implementation has a direct impact on the overall efficiency of cryptographic protocols. In signature schemes and key exchange mechanisms that operate on 256-bit or larger moduli, even moderate improvements in inversion speed translate into measurable performance gains at the protocol level. Consequently, considerable effort has been devoted to optimizing modular inversion algorithms. Two main approaches are commonly used in prime fields. The first relies on exponentiation via Fermat’s little theorem, using the identity $a^{-1} \equiv a^{p-2} \pmod{p}$. This method reduces inversion to a fixed sequence of modular squares and multiplications and is conceptually simple. The second approach is based on the extended Euclidean algorithm and, in particular, on variants of the binary GCD method. Binary-GCD-based algorithms replace general division by shifts, subtractions, and comparisons, often leading to lower arithmetic cost in practice. In recent years, highly optimized inversion algorithms derived from the binary-GCD framework have been developed for prime fields used in practice. These algorithms significantly reduce the number of full modular multiplications and instead rely primarily on additions, subtractions, and shifts. As a result, they achieve superior performance compared to exponentiation-based inversion. However, these optimized algorithms are considerably more intricate than their Fermat-based counterparts.

Beyond performance, modular inversion must satisfy security requirements. In many cryptographic applications, the value being inverted depends on secret material, such as a private key or a per-message nonce. For example, in ECDSA signing, the modular inverse of a secret nonce is computed as part of the signature equation. Leakage of information about this nonce can directly enable recovery of the long-term signing key [18]. Classical Euclidean and binary-GCD algorithms exhibit input-dependent control flow. Their number of iterations varies with the operands, and their execution includes data-dependent branches and early termination conditions. Such behavior can lead to timing and microarchitectural side-channel leakage when the inputs are secret [10, 7]. To mitigate this risk, constant-time variants enforce a fixed number of iterations and eliminate data-dependent branches through masked updates or conditional moves [8]. These transformations preserve the arithmetic structure of the algorithm while ensuring that observable execution characteristics do not depend on secret data. The introduction of a fixed iteration bound, however, complicates the functional correctness proof, because a property previously ensured by the termination condition now has to be verified after the iteration limit is reached.

Pornin’s modular inversion algorithm is a recent example of such a binary-GCD-based, constant-time, high-performance code [16]. What makes it fast is that control-flow decisions are taken based on word-sized approximate values to avoid costly computations, and that the number of iterations is low. The use of approximations makes for loose invariants, which in turn makes the correctness proof especially intricate. It is therefore difficult to be fully confident that the algorithm does not suffer from a bug which would make it susceptible to an attack. In fact, the initial version of Pornin’s paper “used a slightly more aggressive algorithm [...] which was wrong” [16].

This article describes the first mechanized proof of functional correctness of Pornin’s algorithm. To guarantee a high level of confidence, we have formalized the core of the proof using the ROCQ proof assistant [15], and we have then applied it to an imperative description of the algorithm written for the WHY3 verification platform [5]. In doing so, we have filled several holes in the original proof and uncovered a few parts of the proof that were unclear, if not wrong.

The paper is organized as follows. Section 2 gives a succinct description of Pornin’s algorithm and justifies the functional correctness of its optimization layers under sufficient conditions. Section 3 then gives a complete and readable proof of the core soundness

■ **Listing 1** Modular inverse using extended binary GCD.

```

1  function div2 (x m : int) : int =
2    x * ((m + 1)/2)
3
4  function ebin_gcd (y m : int) : int =
5    a ← y, u ← 1, b ← m, v ← 0
6    while a ≠ 0 do
7      if mod a 2 = 0 then
8        (a,u) ← (a/2, div2m u)
9      else
10     if a < b then
11       (a,u,b,v) ← (b,v,a,u)
12     a ← (a - b)/2
13     u ← div2 (u - v) m
14   done;
15   if b ≠ 1 then 0 else v

```

argument, which follows the ROCQ formal proof. Section 4 describes and discusses the WHY3 and ROCQ setup we have used for the mechanical verification, as well as the hurdles we have encountered. Finally, Section 5 gives an overview of the related works and offers some perspectives. When useful, we provide hyperlinks to an online rendering of the ROCQ code.

2 Optimized binary GCD

The modular inverse of y modulo m is a number y^{-1} , such that $y \cdot y^{-1} = 1 \pmod{m}$ holds. When m is a prime number greater than 2, the inverse of a number y , such that $0 \leq y < m$, can be obtained by computing the GCD between y and m along with the associated Bézout coefficients v and u . Indeed, the identity $y \cdot v + m \cdot u = 1$ ensures that $y \cdot v = 1 \pmod{m}$ holds. Listing 1 shows an algorithm based on this approach to compute the modular inverse. The function `div2` computes a number r , such that $x = 2 \cdot r \pmod{m}$. Divisions by two are exact, given the parity of the numbers.

Pornin’s modular inversion algorithm builds on top of Listing 1 while adding three optimization ingredients. First, it replaces the while-loop at Line 6 by a *fixed* amount of iterations. Second, it speeds up the updates on u and v by computing a *transition matrix* and then performing a bulk update after the execution of the loop. Lastly, tests at Line 7 and Line 10 of Listing 1 are performed on *approximated values* of a and b , rather than the original ones. The later approximations are also updated when the Bézout coefficients are.

Listing 2 displays Pornin’s algorithm, which involves two nested for-loops. The outer for-loop iterating over i uses the variables a , b , u , and v in the same way as Listing 1. It computes an approximation $\bar{a}$ (resp. $\bar{b}$) of a (resp. b) that is used to drive the branching in the inner for-loop. These approximations are made by concatenating the $k + 1$ most significant bits and the $k - 1$ least significant bits of the value being approximated. The inner for-loop then performs $k - 1$ iterations of an extended binary GCD algorithm on $\bar{a}$ and $\bar{b}$, using the variables f_0, g_0, f_1, g_1 to record the updates that should be performed after the execution of the loop. At the end of the inner for-loop, the transition matrix is used to perform the bulk update of a, b, u, v . Displayed in green, we show some ghost code that is used to justify some invariants; this is explained in the following subsections.

■ Listing 2 Pornin’s modular inverse algorithm.

```

1  let function PorninGcd (y m k : int)
2    requires { m ≥ 3 }
3    requires { m mod 2 = 1 }
4    requires { 0 ≤ y < m }
5    requires { k > 1 }
6    ensures { GCD(y, m) = 1 → result · y = 1 mod m } =
7    let div2m x = div2 x m in
8    (a, u, b, v) ← (y, 1, m, 0)  (ū, v̄) ← (u, v)
9    for i = 1 to ⌈(2ℓ(m) - 1)/(k - 1)⌉
10     n ← max(ℓ(a), ℓ(b), 2k)
11     ā ← 2k-1⌊a/2n-k-1⌋ + (a mod 2k-1)
12     b̄ ← 2k-1⌊b/2n-k-1⌋ + (b mod 2k-1)
13     (f0, g0, f1, g1) ← (1, 0, 0, 1)  (f̄0, ḡ0, f̄1, ḡ1) ← (f0, g0, f1, g1)
14     for j = 1 to k - 1
15       if mod ā 2 = 0 then
16         ā ← ā/2  (ū, f̄0, ḡ0) ← (div2m ū, div2m f̄0, div2m ḡ0)
17       else
18         if ā < b̄ then
19           (ā, b̄) ← (b̄, ā)  (ū, v̄) ← (v̄, ū)
20           (f0, g0, f1, g1) ← (f1, g1, f0, g0)  (f̄0, ḡ0, f̄1, ḡ1) ← (f̄1, ḡ1, f̄0, ḡ0)
21           ā ← (ā - b̄)/2  ū ← div2m (ū - v̄)
22           (f0, g0) ← (f0 - f1, g0 - g1)  (f̄0, ḡ0) ← (div2m (f̄0 - f̄1), div2m (ḡ0 - ḡ1))
23           (f1, g1) ← (2f1, 2g1)
24         (a, b) ← ((a f0 + b g0)/2k-1, (a f1 + b g1)/2k-1)
25         if a < 0 then
26           (a, f0, g0) ← (-a, -f0, -g0)
27         if b < 0 then
28           (b, f1, g1) ← (-b, -f1, -g1)
29         (u, v) ← ((u f0 + v g0) mod m, (u f1 + v g1) mod m)
30         v ← v/(2(k-1)⌈(2ℓ(m)-1)/(k-1)⌉) mod m
31     if b ≠ 1 then 0 else v

```

To avoid some costly divisions by two when updating the transition matrix, it is implicitly multiplied by two at each inner iteration. So, to compensate for this exponential growth, Line 30 computes $v \cdot 2^{-N} \bmod m$ with $N = (k-1)\lceil(2\ell(m)-1)/(k-1)\rceil$, where $\ell(x)$ denotes the bit length of a positive integer x , and $\lceil \cdot \rceil$ the ceiling function. This N can be efficiently computed by performing a Montgomery reduction. By the way, notice that the number of outer iterations slightly differs from that described in Pornin’s paper. Performing just $\lceil(2\ell(m)-1)/k\rceil$ outer iterations is not enough; it should be $\lceil(2\ell(m)-1)/(k-1)\rceil$. This is more of a typo in Pornin’s algorithm than a deep error, since the corrective factor at Line 30 exhibits the correct cumulative number N of inner iterations. But an inattentive reader would certainly be in for an unpleasant surprise when implementing the algorithm.

Because of the use of the approximations $\bar{a}$ and $\bar{b}$, the comparison test at Line 18 in Listing 2 may not match what would have been obtained if the updated values of a and b had been used, *i.e.*, the test at Line 10 in Listing 1. To recover from this discrepancy, the outer loop adjusts the transition matrix at Line 25 and later to at least make sure that a and b are non-negative, before starting the next iteration.

In the following subsections we gradually construct a functional correctness proof of the algorithm described on Listing 2. We start by studying the extended binary GCD algorithm described in Listing 1. Then, we refine the algorithm by successively introducing the optimization ingredients until reaching the algorithm of Listing 2. For the rest of the section, we thus fix two coprime integers $y, m \in \mathbb{Z}$ such that $3 \leq m$, and $1 \leq y < m$.

2.1 Preliminaries

In this subsection, we prove the functional correctness of the algorithm described in Listing 1. It is ensured when the following set of conditions holds before returning:

1. $a = 0$,
2. $\gcd(y, m) = \gcd(a, b) = 1$,
3. $b = vy \pmod{m}$.

► **Lemma 1.** *Under the above conditions:*

$$y \cdot \text{ebin_gcd}(y, m) = 1 \pmod{m}$$

Proof. Since $a = 0$, we have $b = 1$ because $b = \gcd(0, b) = \gcd(a, b) = \gcd(y, m) = 1$. Thus, $vy = b = 1 \pmod{m}$, which means that the returned value v is the inverse of y . ◀

A proof that the sufficient conditions hold for the extended GCD can be found, for instance, in [6, §1.6.1]. We nonetheless introduce below the required invariants, for the purpose of later describing the impact of the different optimizations at stake.

► **Invariant 2** (a and b stay non-negative). $a, b \geq 0$.

► **Variant 3.** *The value of $a + b$ decreases at every iteration.*

► **Invariant 4.** b is odd.

► **Invariant 5** (The GCD remains constant throughout the execution). $\gcd(y, m) = \gcd(a, b)$

► **Invariant 6.** $a = uy \pmod{m}$ and $b = vy \pmod{m}$.

For the rest of the section, we use Lemma 1 as a blueprint to ensure the functional correctness of every optimization in Pornin's algorithm.

2.2 Fixing the amount of iterations

To verify that fixing the amount of iterations to $2\ell(m) - 1$ is correct in Listing 1, it suffices to apply Lemma 1 and patch the proof that a is 0 when exiting the for-loop. Replacing the while-loop shifts the problem from while-loop termination into a worst-case analysis on the amount of iterations. We use the length in bits as a measure to guarantee that the amount of iterations is enough to reach 0 when exiting the for-loop.

► **Definition 7.** *For any $x \in \mathbb{Z}$, $\ell(x)$ is the bit length of $|x|$, that is, the smallest integer s such that $|x| < 2^s$.*

To guarantee that a is zero, it suffices to show that either (1) a gets to zero in an intermediate iteration, as this fact propagates until the last iteration, or (2) when exiting the for-loop $2\ell(m) - 1$ bits have been lost.

► **Invariant 8.** $a = 0 \vee \ell(y) + \ell(m) \geq \ell(a) + \ell(b) + (i - 1)$

Proof. Assuming a is non-zero, to prove the right-hand side of the disjunction, it suffices to prove that $\ell(a) + \ell(b)$ decreases by one at each iteration of Listing 1. When a is even and non-zero, dividing it by two decreases its bit length by one. (And if a is already 0, it remains 0.) When a is odd, Invariant 2 and Invariant 4 ensure that $\ell(a) + \ell(b)$ decreases, as well as $a \geq b$ thanks to the potential swap before the subtraction. ◀

► **Observation 9.** *If $\ell(y) + \ell(m) \geq \ell(a) + \ell(b) + (2\ell(m) - 1)$ holds at the end of the loop body, then a is zero.*

Proof. By assumption, $2\ell(m) \geq \ell(y) + \ell(m)$. Therefore, $\ell(a) + \ell(b) \leq 1$. Since b is odd by Invariant 4, this forces $\ell(a) = 0$. ◀

► **Corollary 10.** *Fixing the amount of iterations to $2\ell(m) - 1$ ensures the sufficient conditions.*

2.3 Bulk update of the coefficients

To validate the use of nested for-loops and bulk updates, it suffices to show that at least $2\ell(m) - 1$ inner iterations are performed and to update the invariants.

► **Remark 11.** $2\ell(m) - 1 \leq \lceil (2\ell(m) - 1)/(k - 1) \rceil (k - 1)$

To recover an analog of Invariant 6, one should analyze the discrepancy between the updates of coefficients f_0 and f_1 (resp. g_0 and g_1) and that of u and v in Listing 1. In particular, the inner coefficients are wrong by a factor of 2^{j-1} , which makes the outer coefficients wrong by a factor of $2^{(i-1)(k-1)}$. In order to track this error, we introduce a barred ghost variable for each coefficient f_i , g_i , and for u and v , which are updated as in an iteration of Listing 1. To prove the corresponding invariants, we show that every barred coefficient verifies the invariants for the unoptimized GCD, and that every coefficient is congruent to its corresponding barred counterpart, considering an error of factor 2^{j-1} in the inner loops and $2^{(i-1)(k-1)}$ in the outer loops.

The resulting invariants ensure the sufficient conditions, since the factor of two is cancelled by the Montgomery reduction performed at Line 30, thus returning $\bar{v}$, for which $b = \bar{v}y$ holds.

► **Invariant 12.** $a = \bar{u}y \pmod{m}$, $u = 2^{(k-1)(i-1)}\bar{u} \pmod{m}$,
 $b = \bar{v}y \pmod{m}$, $v = 2^{(k-1)(i-1)}\bar{v} \pmod{m}$.

Proof. By symbolic execution. ◀

To guarantee Invariant 12 we show that the following invariant holds.

► **Invariant 13.** $f_0 = 2^{j-1}\bar{f}_0 \pmod{m}$, $f_1 = 2^{j-1}\bar{f}_1 \pmod{m}$,
 $g_0 = 2^{j-1}\bar{g}_0 \pmod{m}$, $g_1 = 2^{j-1}\bar{g}_1 \pmod{m}$.

Proof. By symbolic execution. ◀

► **Corollary 14.** *Performing bulk updates ensures the sufficient conditions.*

2.4 Approximating the input values

We now make explicit the invariants that justify the use of approximations. We first fix some terminology for referring to the state of the different variables over the execution of an inner loop, for an arbitrary fixed iteration of the outer loop.

We refer to *step* t of the inner loop, for $t = 0 \dots k - 1$, for the state of the program *after* t iterations. In particular, step $t = k - 1$ refers to its state *after* all iterations have been completed. Conversely, step $t = 0$ refers to the state of the program *before* the execution of the loop. More generally, for $t = 0 \dots k - 2$, step t corresponds to the state of the program before entering iteration $j = t + 1$.

For x among $\bar{a}$, $\bar{b}$, f_0 , f_1 , g_0 , g_1 and for any $t = 0 \dots k - 1$, we write x_t for the value of variable x at step t . Yet, we sometimes casually write x instead of x_0 , for the sake of readability. For instance,

$$\begin{aligned}\bar{a} &= \bar{a}_0 = (a_0 \bmod 2^{k-1}) + 2^{k-1} \lfloor a_0 / 2^{n-k-1} \rfloor, \\ \bar{b} &= \bar{b}_0 = (b_0 \bmod 2^{k-1}) + 2^{k-1} \lfloor b_0 / 2^{n-k-1} \rfloor.\end{aligned}$$

For the sake of the analysis, it is useful to also consider the following auxiliary variables:

$$a_t \triangleq (af_{0,t} + bg_{0,t})/2^t, \quad b_t \triangleq (af_{1,t} + bg_{1,t})/2^t.$$

Note that a_0 (resp. b_0) coincides with a (resp. b) and a_{k-1} (resp. b_{k-1}) coincides with the assignment at Line 24 of Listing 2.

Crucially, a_t and b_t have the same parity as their respective approximations.

► **Lemma 15.** *For any $t = 0 \dots k - 1$, we have $2^t \bar{a}_t = \bar{a} f_{0,t} + \bar{b} g_{0,t}$ and $2^t \bar{b}_t = \bar{a} f_{1,t} + \bar{b} g_{1,t}$. Moreover, $\bar{b}$ is odd, and at any step $t = 0 \dots k - 2$, a_t and $\bar{a}_t$ have the same parity, while b_t and $\bar{b}_t$ are odd.*

Proof. Identities are proved by induction on t ; corollaries follow from elementary modular arithmetic. ◀

To verify that using approximations is correct, it suffices to show that the following properties hold when exiting the inner loop:

1. $\ell(a_{k-1}) + \ell(b_{k-1}) \leq \ell(a) + \ell(b) - (k - 1)$,
2. $\gcd(a, b) = \gcd(a_{k-1}, b_{k-1})$,
3. $b_{k-1} = \bar{v}y \pmod{m}$.

These properties are actual invariants for the outer loop. Note that they are not affected by the corrections performed between Line 25 and Line 29, *i.e.*, changing the sign of the values to make them non-negative; nor by having performed a misplaced swap when both $\bar{a}$ and $\bar{b}$ are odd. In particular, properties 2 and 3 are available for $\bar{a}$ and $\bar{b}$ by Corollary 14 and Invariant 5, but their proof only relies on the parity of the values driving the branching structure of the iteration. Thus, by Lemma 15 we can preserve properties 2 and 3 as inner loop invariants, in the following sense:

► **Invariant 16.** $\gcd(a, b) = \gcd(a_t, b_t)$.

► **Invariant 17.** $a_t = \bar{u}y \pmod{m}$, $b_t = \bar{v}y \pmod{m}$.

If either $a_{k-1} = 0$ or property 1 holds, then the outer loop performs enough bit-length reductions to ensure $a = 0$.

► **Invariant 18.** *If the inner loop ensures $a_{k-1} = 0$ or if property 1 holds after executing an inner loop with index i , then $a = 0 \vee \ell(y) + \ell(m) \geq \ell(a) + \ell(b) + (i - 1)(k - 1)$.*

► **Lemma 19.** *Property 1 ensures sufficient conditions for Listing 2.*

We dedicate the following section to the proof of the first property.

3 Bit reduction of the inner loop

In this section, we consider a fixed arbitrary iteration of the outer loop and study the corresponding execution of the inner loop, which itself involves $k - 1$ iterations.

 **Theorem 20.** *If $a_{k-1} \neq 0$ then*

$$\ell(a_{k-1}) + \ell(b_{k-1}) \leq \ell(a) + \ell(b) - (k - 1).$$

The rest of the section is devoted to prove Theorem 20. We provide a self-contained account of the proof, which is significantly different from that sketched in the Appendices of Pornin's note [16], albeit broadly similar in structure. We refer to the latter when possible, but provide an alternative presentation when we believe it is relevant.

3.1 Preliminaries

Before describing the actual proof, we introduce some vocabulary and notations, and we state some useful elementary facts. First, we introduce a taxonomy of states, according to the branch of the algorithm taken at the corresponding iteration, and to its effect on the sum of bit lengths.

► **Definition 21.** *A step t is halving if $\bar{a}_t$ is even. A step t is a swap if $\bar{a}_t$ is odd and $\bar{a}_t < \bar{b}_t$. A non-halving step t (i.e., such that $\bar{a}_t$ is odd) is divergent if*

$$(\bar{a}_t < \bar{b}_t \wedge a_t \geq b_t) \vee (\bar{a}_t \geq \bar{b}_t \wedge a_t < b_t).$$

Otherwise, it is a synchronous step.

► **Definition 22.** *For any $t = 0 \dots k - 1$, we say that step t is*

- decreasing if $\ell(a_{t+1}) + \ell(b_{t+1}) \leq \ell(a_t) + \ell(b_t) - 1$ (and $t \leq k - 2$),
- regular if $\ell(a_t) + \ell(b_t) \leq \ell(a_0) + \ell(b_0) - t$,
- good if $\ell(a_t) + \ell(b_t) \leq \ell(a_0) + \ell(b_0) - (k - 1)$.

Moreover, t is a lasting good (resp. lasting decreasing, resp. lasting regular) step if for any $t' \geq t$, step t' is good (resp. decreasing, resp. regular).

► **Remark 23.** Theorem 20 states that step $t = k - 1$ is good, or, equivalently, regular.

 **Lemma 24.** *If at step t and for some integer $\alpha \geq 0$ both $|a_t| < 2^\alpha$ and $|b_t| < 2^\alpha$, then for any subsequent step $u \geq t$ we also have $|a_u| < 2^\alpha$ and $|b_u| < 2^\alpha$. In particular, if $2\alpha \leq \ell(a) + \ell(b) - (k - 1)$, then t is a lasting good step.*

Proof. By induction on t , observing that if $|a_u| < 2^\alpha$ and $|b_u| < 2^\alpha$, then

$$|(a_u - b_u)/2| \leq (|a_u| + |b_u|)/2 < 2^\alpha. \quad \blacktriangleleft$$

In particular, synchronous steps are decreasing, as the execution proceeds as in the non-optimized variant.

 **Lemma 25.** *A synchronous step $t \leq k - 2$ with $a_t \neq 0$ is decreasing.*

Proof. Let t be a synchronous step with $a_t \neq 0$. By Lemma 15, a_t (resp. b_t) and $\bar{a}_t$ (resp. $\bar{b}_t$) have the same parity. Then either t is halving and

$$a_{t+1} = a_t/2, \quad b_{t+1} = b_t$$

and since a_t is even and non-zero, t is decreasing. Otherwise, if t is not a swap step,

$$b_{t+1} = b_t, \quad \bar{b}_{t+1} = \bar{b}_t, \quad \bar{a}_{t+1} = (\bar{a}_t - \bar{b}_t)/2, \quad a_{t+1} = (a_t - b_t)/2.$$

In this case, a_t and b_t are both odd so $\ell(a_{t+1}) \leq \ell(a_t) - 1$ and t is thus decreasing. If t is a swap step, then similarly

$$\ell(b_{t+1}) = \ell(a_t), \quad \ell(a_{t+1}) \leq \ell(b_t) - 1. \quad \blacktriangleleft$$

 **Lemma 26.** *For any step t with $0 \leq t \leq k-1$, we have*

$$|f_{0,t}| + |g_{0,t}| \leq 2^t, \quad |f_{1,t}| + |g_{1,t}| \leq 2^t.$$

Proof. Cf Pornin's Appendix A1. $\blacktriangleleft$

 **Lemma 27.** *For any step t with $0 \leq t \leq k-1$, we have*

$$|a_t - 2^{n-2k}\bar{a}_t| \leq 2^{n-k-1}, \quad |b_t - 2^{n-2k}\bar{b}_t| \leq 2^{n-k-1},$$

and the inequality is strict at $t = 0$.

Proof. We prove the result for a_t , the case of b_t being similar. For $t = 0$, define

$$r_1 \triangleq a \pmod{2^{n-k-1}}, \quad r'_1 \triangleq a \pmod{2^{k-1}}, \quad r_2 = \lfloor a/2^{n-k-1} \rfloor,$$

so that

$$\begin{aligned} a &= r_1 + 2^{n-k-1}r_2 && \text{with } 0 \leq r_1 < 2^{n-k-1} \\ \bar{a} &= r'_1 + 2^{k-1}r_2 && \text{with } 0 \leq r'_1 < 2^{k-1} \end{aligned}$$

and thus

$$a - 2^{n-2k}\bar{a} = r_1 - 2^{n-2k}r'_1.$$

Now since $0 \leq r_1 < 2^{n-k-1}$ and $0 \leq 2^{n-2k}r'_1 < 2^{n-k-1}$, we have $|r_1 - 2^{2k-n}r'_1| < 2^{n-k-1}$. For any $t = 1 \dots k-1$, we follow Pornin and write:

$$\begin{aligned} a_t - 2^{n-2k}\bar{a}_t &= 2^{-t}[(af_{0,t} + bg_{0,t}) - 2^{n-2k}(\bar{a}f_{0,t} + \bar{b}g_{0,t})] \\ &= 2^{-t}[(a - 2^{n-2k}\bar{a})f_{0,t} + (b - 2^{n-2k}\bar{b})g_{0,t}]. \end{aligned}$$

We then successively use triangular inequalities, case $t = 0$, and Lemma 26 to conclude:

$$\begin{aligned} |a_t - 2^{n-2k}\bar{a}_t| &\leq 2^{-t}(|a - 2^{n-2k}\bar{a}||f_{0,t}| + |b - 2^{n-2k}\bar{b}||g_{0,t}|) \\ &\leq 2^{n-k-1-t}(|f_{0,t}| + |g_{0,t}|) \leq 2^{n-k-1}. \quad \blacktriangleleft \end{aligned}$$

 **Lemma 28.** *For a first divergence at step d , we have $|a_{d+1}| \leq 2^{n-k-1}$ and $|b_d - a_d| \leq 2^{n-k}$. Moreover, $|b_d - a_d| = 2^{n-k}$ implies $\bar{b}_d = \bar{a}_d$.*

Proof. We detail the proof in the case of $a_d < b_d$ and $\bar{a}_d \geq \bar{b}_d$. The other case, $a_d \geq b_d$ and $\bar{a}_d < \bar{b}_d$, is similar.

Then $a_{d+1} = (a_d - b_d)/2$. We follow the proof in Pornin's Appendix A1 and write

$$2^{n-2k}(\bar{a}_d - \bar{b}_d) - (a_d - b_d) = (2^{n-2k}\bar{a}_d - a_d) - (2^{n-2k}\bar{b}_d - b_d)$$

and thus by Lemma 27

$$|2^{n-2k}(\bar{a}_d - \bar{b}_d) - (a_d - b_d)| \leq |2^{n-2k}\bar{a}_d - a_d| + |2^{n-2k}\bar{b}_d - b_d| \leq 2^{n-k}.$$

Now observe that $2^{n-2k}(\bar{a}_d - \bar{b}_d) \geq 0$ and $a_d - b_d < 0$, which gives

$$|2^{n-2k}(\bar{a}_d - \bar{b}_d) - (a_d - b_d)| = 2^{n-2k}(\bar{a}_d - \bar{b}_d) + (b_d - a_d).$$

Thus, $0 \leq \bar{a}_d - \bar{b}_d < 2^k$ and $0 < b_d - a_d \leq 2^{n-k}$.

Moreover, if $b_d - a_d = 2^{n-k}$, then $2^{n-2k}(\bar{a}_d - \bar{b}_d) = 0$ (since their sum is bounded by 2^{n-k}), and thus $\bar{a}_d = \bar{b}_d$. This concludes the proof in this case. ◀

3.2 Divergence-free case

We first consider the case of a and b having initial values that are distant enough. In this case, each step in the execution of the loop is decreasing, and divergence never occurs. Lemma 30 provides a sufficient condition ensuring a block of consecutive decreasing steps.

► **Observation 29.** *Synchronous swap steps and halving steps are decreasing.*

🔥 **Lemma 30.** *Consider a step $0 \leq s \leq k-2$ such that*

$$0 < a_s, \quad 0 < \bar{a}_s, \quad 0 < b_s, \quad 0 < \bar{b}_s, \quad k-s \leq \ell(a_s), \quad k-s \leq \ell(\bar{a}_s).$$

Suppose that for some $2 \leq u \leq k-s$ we have

$$\ell(b_s) \leq \ell(a_s) - u, \quad \ell(\bar{b}_s) \leq \ell(\bar{a}_s) - u.$$

Then any step $s+t$ with $t \leq u-2$ is decreasing.

Proof. It suffices to prove that for any $0 \leq t \leq u-2$ we have

$$b_{s+t} = b_{s+t+1} = b_s, \quad 0 < a_{s+t}, \quad \text{and step } s+t \text{ is decreasing.} \quad (1)$$

We actually first establish that the following conjunction implies property (1):

$$a_{s+t} \geq (a_s - b_s(2^t - 1))/2^t, \quad (2)$$

$$\bar{a}_{s+t} \geq (\bar{a}_s - \bar{b}_s(2^t - 1))/2^t, \quad (3)$$

$$b_{s+t} = b_s \text{ and } \bar{b}_{s+t} = \bar{b}_s. \quad (4)$$

Indeed, properties (2) and (3) respectively entail that

$$a_{s+t} \geq 2^{\ell(a_s)-2-t} > 0, \quad \bar{a}_{s+t} \geq 2^{\ell(\bar{a}_s)-2-t}.$$

Now combined with (4), this also entails that $a_{s+t} \geq b_{s+t} = b_s$ and $\bar{a}_{s+t} \geq \bar{b}_{s+t} = \bar{b}_s$ so step $s+t$ is a synchronous swap, which in turn implies (1). Let us now simultaneously prove the conjunction of (2), (3), and (4) by induction on t . The base case is immediate. Suppose that (2), (3), and (4) hold at step t . If $a_{t+1} = a_t/2$ and $\bar{a}_{t+1} = \bar{a}_t/2$, we conclude by an easy calculation. Otherwise, we have

$$a_{s+t+1} = (a_{s+t} - b_{s+t})/2 \geq (a_s - b_s(2^t - 1))/2^{t+1} - b_s/2 \geq (a_s - b_s(2^{t+1} - 1))/2^{t+1}.$$

A similar calculation proves that $\bar{a}_{s+t+1} \geq (\bar{a}_s - \bar{b}_s(2^{t+1} - 1))/2^{t+1}$. As observed above, (2), (3) and (4) entail that step $(s+t)$ is a synchronous swap, thus decreasing by Lemma 25. ◀

🔴 **Lemma 31.** *If $a > 0$ with $\ell(a) = n$ and $\ell(b) \leq n - k$, then step $t = 0$ is lasting decreasing.*

Proof. Note that $\ell(b) \leq n - k$ implies that $b < 2^{n-k}$. Therefore, $\lfloor b/2^{n-k-1} \rfloor < 2$ and $\bar{b} < 2^k$. Similarly, $\lfloor a/2^{n-k-1} \rfloor \geq 2^k$ so $\bar{a} \geq 2^{2k-1}$. Lemma 30 thus applies with $s = 0$ and $u = k$. ◀

► **Lemma 32.** *If $a > 0$ with $\ell(b) = n$ and $\ell(a) \leq n - k$, then either $a_{k-1} = 0$ or step $t = 0$ is lasting decreasing.*

Proof. As in the previous case, observe that $\bar{a}_0 = \bar{a} < 2^k$ and $\bar{b}_0 = \bar{b} \geq 2^{2k-1}$. If a_0 is odd, then so is $\bar{a}_0$. Since $\bar{a}_0 < \bar{b}_0$, a swap occurs at t_0 . Since moreover $a_0 < b_0$, the execution behaves as if a and b had been swapped in the initial conditions and Lemma 31 applies.

If $\bar{a}_0$ is even, then let $s \triangleq \min\{r < k-1 \text{ such that } r \text{ is not halving}\}$. For any $0 \leq t \leq s-1$, $a_{t+1} = a_t/2$ and $b_{t+1} = b_t$ so t is decreasing. If $s = k-2$ this suffices to conclude. If $s < k-1$, then for $t = s$ we have

$$a_s = a/2^s, \quad \bar{a}_s = \bar{a}/2^s, \quad b_s = b, \quad \bar{b}_s = \bar{b}.$$

In particular $a_s < b_s$ and $\bar{a}_s < \bar{b}_s$ so a swap happens. Right after this swap, we have (slightly abusing the x_s notation)

$$a_s = b, \quad \bar{a}_s = \bar{b}, \quad b_s = a/2^s, \quad \bar{b}_s = \bar{a}/2^s,$$

and in particular

$$\bar{a}_s \geq 2^{2k-1}, \quad 0 \leq b_s < 2^{n-k-s}, \quad \bar{b}_s < 2^{k-s}.$$

Therefore,

$$\ell(a_s) = n, \quad \ell(\bar{a}_s) \geq 2k, \quad \ell(b_s) \leq n - k - s, \quad \ell(\bar{b}_s) \leq k - s.$$

If $a_s = 0$ then $a_{k-1} = 0$. Otherwise, Lemma 30 with $s = s$ and $u = k - s$ proves that for $s \leq t \leq k - 2$, step t is decreasing and this way concludes the proof. ◀

► **Remark 33.** A variant of Lemma 30 is implicit in Pornin's Appendix A2. Its mechanized proof, however, requires some care, not only to sufficiently strengthen the induction hypothesis, but also to accommodate its usage in the proof of Lemma 32.

🔴 **Theorem 34.** *When $a > 0$, if either $\ell(b) = n$ and $\ell(a) \leq n - k$, or $\ell(a) = n$ and $\ell(b) \leq n - k$, then either $a_{k-1} = 0$ or step $k - 1$ is good.*

Proof. Direct corollary of Lemma 31 and Lemma 32. ◀

3.3 Divergence case

We now consider executions that possibly involve a divergent step. The proof detailed below fills several holes in the original writing. Throughout this subsection we suppose that $\ell(a) = n$ and $\ell(b) > n - k$ or $\ell(b) = n$ and $\ell(a) > n - k$, so that $\ell(a) + \ell(b) \geq 2n - k + 1$. We also assume $a > 0$. We will make use of the following elementary but crucial observations.

► **Observation 35.** *Suppose $\ell(a) + \ell(b) \geq 2n - k + 1$. Then for any t , if both $|a_t| < 2^{n-k+1}$ and $|b_t| < 2^{n-k+1}$, step t is lasting good.*

Proof. Immediate by Lemma 24, with $\alpha = n - k + 1$ because $1 \leq k$. ◀

► **Observation 36.** *Let d be the first divergence step of the execution, and take $d = k - 1$ if there is no such step. Then for any step t such that $0 \leq t \leq d$, we have $\min(a_t, b_t) \leq \min(a, b)$.*

Proof. By induction on t . ◀

► **Observation 37.** *Let d be the first divergence step of the execution, and take $d = k - 1$ if there is no such step. Then for any step t such that $0 \leq t \leq d$, we have $0 \leq a_t$ and $0 \leq b_t$.*

Proof. By induction on t (and symbolic execution of an iteration). ◀

Under the fixed assumptions of this section, we now prove Theorem 20 using the following strategy: We consider the first divergent step d , if any, and prove that step $d + 1$ is good. Then, we introduce a certain disjunctive predicate H and establish that H is an invariant from $d + 1$ on, while also preserving good steps. As a consequence, $d + 1$ is a lasting good step, which allows us to conclude. First, let us define

$$d \triangleq \min(\{t \mid t \text{ is neither halving nor synchronous}\} \cup \{k - 1\}).$$

For any step t with $0 \leq t \leq d - 1$, step t is either halving or synchronous and hence decreasing. Therefore d is regular. If $d = k - 1$, then Theorem 20 follows. Otherwise, remember that $0 \leq b_d$ and $0 \leq a_d$ after Observation 37. Pose $M_t \triangleq \max(a_t, b_t)$ and $m_d \triangleq \min(a_t, b_t)$. If $M_d < 2^{n-k+1}$, step d is lasting good by Observation 35. Otherwise, by Lemma 28 we have

$$|m_d - M_d| = |b_d - a_d| \leq 2^{n-k}.$$

Because step d is divergent, we have

$$b_{d+1} = M_d \geq 0 \quad \text{and} \quad a_{d+1} = (m_d - M_d)/2 \leq 0.$$

In particular $|a_{d+1}| = |a_d - b_d|/2 \leq 2^{n-k-1}$ and $\ell(a_{d+1}) \leq n - k$.

We now prove that step $d + 1$ is a good step. Pose $\ell(b_d) = n - k + \lambda$ and $\ell(a_d) = n - k + \mu$. In particular $\ell(M_d) = n - k + \max(\lambda, \mu)$ and $\ell(m_d) = n - k + \min(\lambda, \mu)$. Remember that we have $M_d \geq 2^{n-k+1}$ and therefore $|m_d - M_d| \leq M_d/2$ and $M_d \leq 2m_d$. As a consequence,

$$|\ell(M_d) - \ell(m_d)| \leq 1 \quad \text{and thus} \quad |\lambda - \mu| \leq 1.$$

Observe that $\ell(m_d) = \min(\ell(a_d), \ell(b_d))$, because step d is the first divergent step by assumption. Following Observation 36, we thus have

$$\begin{aligned} \ell(a_d) + \ell(b_d) &= \ell(m_d) + \ell(M_d) \\ &= \ell(m_d) + (n - k + \max(\lambda, \mu)) \\ &\leq \min(\ell(a), \ell(b)) + (n - k + \max(\lambda, \mu)) \\ &\leq \ell(a) + \ell(b) - k + \max(\lambda, \mu). \end{aligned}$$

Moreover, we can write $n - k = \ell(m_d) - \min(\lambda, \mu) = \min(\ell(a_d), \ell(b_d)) - \min(\lambda, \mu)$, so that

$$\begin{aligned} \ell(a_{d+1}) + \ell(b_{d+1}) &= \ell(a_{d+1}) + \ell(M_d) \\ &\leq \min(\ell(a_d), \ell(b_d)) - \min(\lambda, \mu) + \max(\ell(a_d), \ell(b_d)) \\ &\leq \ell(a_d) + \ell(b_d) - \min(\lambda, \mu) \\ &\leq \ell(a) + \ell(b) - k + \max(\lambda, \mu) - \min(\lambda, \mu) \\ &\leq \ell(a) + \ell(b) - k + 1. \end{aligned}$$

So step $d + 1$ is a good step. We now prove that it is lasting good. For a step t with $0 \leq t \leq k - 2$, define

$$P(t) \triangleq M_t \geq 2^{n-k+1} \wedge -2^{n-k} < m_t \leq 0, \quad Q(t) \triangleq \max(|a_t|, |b_t|) < 2^{n-k+1}$$

and pose $H(t) \triangleq P(t) \vee Q(t)$.

 **Observation 38.** *Predicate H holds at step $d + 1$.*

Proof. Remember that $b_{d+1} = M_d \geq 0$ and $a_{d+1} = (m_d - M_d)/2 \leq 0$. Hence $m_{d+1} = a_{d+1}$ and $M_{d+1} = b_{d+1}$. Moreover, $|a_{d+1}| < 2^{n-k}$, so either P or Q holds at $d + 1$ depending on whether $M_{d+1} \geq 2^{n-k+1}$ holds or not. $\blacktriangleleft$

 **Lemma 39.** *If Q holds at step t , then Q holds at $t + 1$. Moreover, step t is lasting good.*

Proof. Direct by Observation 35. $\blacktriangleleft$

 **Lemma 40.** *If P holds at step t , then H holds at step $t + 1$. Moreover, if $a_t \neq 0$,*

$$\ell(a_{t+1}) + \ell(b_{t+1}) \leq \ell(a_t) + \ell(b_t).$$

Proof. Suppose that P holds at step t . If step t is halving, then

$$\ell(a_{t+1}) + \ell(b_{t+1}) \leq \ell(a_t) + \ell(b_t).$$

If it halves m_t , then P holds at step $t + 1$. If it halves M_{t+1} and $M_t \geq 2^{n-k+2}$, then P also holds at step $t + 1$. If $M_t < 2^{n-k+2}$, then $M_{t+1} < 2^{n-k+1}$ and Q holds at step $t + 1$. If step t is not halving then observe that

$$2^{n-2k}(\bar{b}_t - \bar{a}_t) = (2^{n-2k}\bar{b}_t - b_t) + (b_t - a_t) - (2^{n-2k}\bar{a}_t - a_t)$$

with $|2^{n-2k}\bar{b}_t - b_t| + |2^{n-2k}\bar{a}_t - a_t| \leq 2^{n-k}$ by Lemma 27. Since P entails $|b_t - a_t| > 2^{n-k}$, this proves that $b_t - a_t$ and $\bar{b}_t - \bar{a}_t$ are both strictly positive, or both strictly negative, from which it follows that t is synchronous. Therefore,

$$b_{t+1} = m_t \leq 0 \quad \text{and} \quad a_{t+1} = (M_t - m_t)/2 \geq 0$$

Moreover,

$$m_{t+1} = b_{t+1} \leq 0, \quad |m_{t+1}| \leq 2^{n-k}, \quad M_{t+1} = a_{t+1} \geq 0,$$

so H holds at step $t + 1$. In addition, $|a_{t+1}| \leq (|M_t| + |m_t|)/2 \leq |M_t|$, so

$$\ell(a_{t+1}) + \ell(b_{t+1}) \leq \ell(M_t) + \ell(m_t) = \ell(a_t) + \ell(b_t). \quad \blacktriangleleft$$

 **Corollary 41.** *If H holds at a good step t then step t is lasting good.*

Proof. Suppose that H holds at a good step t . We prove by induction on t that at any step $t' \geq t$, H holds and t' is good. Lemma 39 and Lemma 40 prove that H holds at $t + 1$. If Q holds at $t + 1$ then step $t + 1$ is (lasting) good by Lemma 39. If P holds at $t + 1$ then Lemma 40 ensures that step $t + 1$ is good as well. $\blacktriangleleft$

$\blacktriangleright$ **Theorem 42.** *If step d is the first divergent step, then step d is lasting good.*

Proof. Since step d is regular, if $d = k - 1$, d is (lasting) good. Otherwise, $d + 1$ is good and H holds at $d + 1$. Therefore, step $d + 1$ is lasting good by Corollary 41, and thus so is d . $\blacktriangleleft$

4 Mechanized proofs

We now turn to the proof-engineering aspects of this endeavor. In Section 4.1 we explain how we used WHY3 to specify Listing 2, and to mechanize the content of Section 2. Then, Section 4.2 discusses how the proof of Section 3 was conducted using the ROCQ interactive prover. Finally, we offer some qualitative and quantitative comments on this mechanized proof in Section 4.3.

4.1 Deductive verification in Why3

WHY3 is a platform for the deductive verification of programs [5]. While it can be used as a back-end for various verification systems for mainstream programming languages, it also provides its own programming and specification language, WhyML, featuring mutable variables, loops, etc. We used WhyML to describe Pornin’s algorithm before proving its functional correctness essentially following the proof described in Section 2. WHY3, however, does not provide any specific support for proofs via refinements, which is only reflected in the proof engineering methodology. The latter consisted in gradually modifying the code of the algorithm, so as to successively introduce each of the three optimizations and, at each stage, to repair the existing invariants broken by the modification. This process eventually led to the program displayed in Listing 2, with an extra postcondition to describe the behavior when the inputs are not coprime. The actual specification, however, is much more verbose, as the program is annotated with numerous clauses: preconditions, postconditions, loop invariants, assertions, and crucially, ghost code.

Ghost code makes it possible to reason on the values that a and b would have had if their update had not been delayed. That is, we can keep track of the value assigned to the auxiliary variables a_t and b_t defined in Section 2.4, and prove, for instance, that their parity coincides with the control-flow decisions driven by the parity of approximations, as described in Lemma 15. Ghost code is also used to track divergence from the classical algorithm, and to state and prove invariants about the potential post-divergence iterations. Proving these invariants entails that, if the arguments of Pornin’s modular inverse algorithm satisfy the preconditions, then its result satisfies the postcondition. This implication is itself entailed by the weakest precondition, which is generated as a verification condition by WHY3. The proof then proceeds by gradually massaging verification conditions until they are discharged by external automated provers.

The present proof relies on the automated solvers Alt-Ergo, CVC5, and Z3. Unfortunately, SMT solvers often failed to prove the numerous straightforward but non-linear statements involved in verification conditions. We therefore had to include a number of extra hints to the WhyML code, so as to guide the SMT solvers toward a proof. Even so, SMT solvers were completely unable to discharge a handful of verification conditions and it just proved more efficient to resort to exporting verification conditions as ROCQ statements. For instance, Theorem 20 is completely out of reach for SMT solvers.

For this export, we did not use the standard ROCQ output of WHY3, but rather wrote a custom driver file¹ which maps a few symbols found in the WHY3 theories to the corresponding symbols in the MATHCOMP libraries [12]. The latter indeed features useful libraries for reasoning about elementary number theory. The exhaustive list of mapped symbols is as follows: the basic arithmetic operations over integers ($+$, $-$, $\times$, $\leq$, $<$), the power function, the

¹ Available at: `driver.html`

Euclidean division and modulo, and the GCD function. This driver file should be considered as an axiom, as any error in it would endanger the consistency of the whole development. That is why we have kept the driver file as small as possible.

4.2 Deductive verification in Rocq

The ROCQ file exported by WHY3 only contains a theorem statement, as well as the context needed to express it; the proof then has to be provided manually.² Yet the actual formalization of the proof of Theorem 20, namely the content of Section 3, was actually developed in a separate ROCQ file, independently from this export. The final proof thus consists in justifying that the definitions exported from WHY3 to ROCQ align with the ROCQ definitions used by that of Theorem 20.

In fact, the actual mechanized proof of Theorem 20 can unfortunately hardly reuse the context exported by WHY3, which explains this possibly surprising architecture of the proof. Indeed, a number of useful facts, typically those mentioned in Section 3.1, are only provided for a *fixed* iteration, while we actually need their *universally quantified* versions, typically when reasoning by induction. The corresponding proofs thus have to be duplicated in ROCQ, but without the support for automation WHY3 offers. On the other hand, mechanizing the two-step induction featured by the proof of Lemma 30, or the abuse of notation in that of Lemma 32 causes no specific technical difficulties in ROCQ. A significant pain point, however, remains the limited automation available for non-linear arithmetic, in ROCQ as well. Moreover, the resulting pedestrian proofs are further hampered by the numerous casts needed to navigate between the types of natural numbers, integers, and bounded integers. This known issue actually motivated the development of specific automation tools in the Mathlib library [11]. Note that this problem is mostly irrelevant for goals pertaining to linear arithmetic, thanks to the recent improvements brought to the corresponding tactics [17].

4.3 Facts and figures

We now provide some qualitative and quantitative remarks on the mechanization. Conducting the proof in WHY3, as opposed to a general-purpose interactive prover like ROCQ, allowed us to describe the algorithm in a form much closer to its original presentation than it would be using ROCQ as a programming language. This proof of functional correctness crucially relies on both the automation provided by WHY3 and on the ability to conduct in ROCQ the parts of the reasoning that do not strictly follow the execution flow of the algorithm. As mentioned in Section 4.2, this interaction, however, comes at the price of duplicating some proofs on the ROCQ side.

For instance, the WHY3 proof of Lemma 27 depends on 6 invariants, of which 4 need to be proven simultaneously, *i.e.*, as a conjunction. Yet on the ROCQ side, there is no analog to the weakest precondition expression automatically generated by WHY3. Instead, the proof resorts to equational lemmas like the following one:

```
(* values of a after an iteration on an initial state vs with odd a *)
Lemma a_odd_s (vs : iter_vars) :
  ¬ (is_halving vs) →
  a (inner_iter vs) =
  a (if (ā vs) < (b̄ vs) then odd_not_swap (swap vs) else odd_not_swap vs).
```

² Link to this statement: why3VC.html#ebin_gcd'vc

■ **Table 1** Sizes of the different components of the proof.

Nº of	Invariants	Lemmas	Interactive proofs	LOC	VC
WHY3	71	259	66	3003	1649
Rocq GCD	–	293	–	4195	–
Rocq VC	–	79	–	4361	–

where $(\text{vs} : \text{iter_vars})$ is a tuple $(a, b, \bar{a}, \bar{b}, f_0, g_0, f_1, g_1)$ storing the values of all the variables used in the program, and (x vs) extracts the value of variable x in this tuple. Function `inner_iter` corresponds to the execution of a single iteration on a given tuple of variables, which itself calls the auxiliary function `odd_not_swap` in the odd and non-swapping case. Finally, `(swap vs)` just swaps the values of $(\bar{b} \text{ vs})$ and $(\bar{a} \text{ vs})$ in vs . Similar ancillary lemmas are proven for every branch in Listing 2 and for every variable of the tuple, and eventually used to control the unfolding of `inner_iter` in subsequent proofs.

On the other hand, the WHY3 part of the proof requires finding a balance between meaningful proof steps, and the possibly redundant facts that SMT solvers require accumulating in order to eventually discharge all proof obligations. As already mentioned in Section 4.1, the pervasive use of modular and non-linear arithmetic unfortunately tilted this balance towards guiding the SMT solvers. The large number of extra assertions that were needed negatively impacts the readability of the code, and induces an explosion in size of the logical context passed to SMT solvers, which significantly slows down the proof search. This forced us to systematically encapsulate each proof as a let-lemma with minimized pre- and post-conditions. This way, all the ad-hoc hints that are added to help discharge a proof obligation at least do not pollute the logical context of later lemmas.

Table 1 provides a quantitative description of the respective components of the final proof. Rocq GCD stands for the mechanization of the content in Section 3, and Rocq VC for the verification condition delegated to Rocq by WHY3. The latter is split into 3632 lines to specify the statement, and 728 lines for the proof connecting definitions in WHY3 with the ones of Rocq GCD.

5 Conclusion

This paper describes the first mechanized proof of correctness of Pornin’s algorithm. In doing so, we have filled numerous holes in Pornin’s original proof, and we can now confidently state that the algorithm is no longer just “believed to be correct” [16], but that it *is* correct.

Despite their critical importance for some cryptography protocols, we are only aware of a few attempts at mechanically checking the implementation of a state-of-the-art modular inversion algorithm (or equivalently, an extended GCD algorithm). A noteworthy example is the Safegcd algorithm, proposed by Bernstein and Yang in 2019 [4]. This algorithm looks deceptively simple as it is just the binary GCD algorithm, except that the decision to swap the integers comes from an auxiliary variable rather than from the integers themselves. Since the decision is independent of the large integers, the algorithm can be optimized in a way similar to Pornin’s, that is, an inner loop emulates a sequence of reductions by computing a transition matrix, using only machine-word approximations of the integers.

Therefore, as with Pornin’s algorithm, the crux of the correctness proof is to guarantee that sufficiently many iterations were performed, despite the possibly wrong guesses. To the best of our knowledge, there have been two formal proofs of this result. One is by Bernstein himself; he wrote a SageMath script that outputs a proof for the HOL Light system [3].

The other one is by O'Connor and Poelstra; not only did they prove with the ROCQ proof assistant the correctness of the algorithm, they also proved a C implementation using the Verified Software Toolchain [14].

The s2n-bignum library initiated by John Harrison [9] contains state-of-the-art implementations of numerous cryptography primitives. They are written in x86-64 and ARM assembly, and their decompiled machine code is verified using HOL Light. In particular, the correctness proofs for the inversion algorithms for specific moduli rely on Bernstein's proof generator for proving that they execute enough iterations. More interestingly, when the moduli are generic, the s2n-bignum library contains an inversion algorithm that looks very similar to Pornin's one.³ The implementation is a bit less intricate though (*e.g.*, the high and low parts are kept in separate registers, instead of being combined as in Pornin's algorithm). Indeed, implementing the unadulterated algorithm was not deemed worth the formalization effort, given the concerns about the pen-and-paper proof (personal communication). Despite the simpler algorithm, there are some similarities with our work in the way the proof of correctness is performed. In particular, in both works, the invariant of the inner loop is quite large and it features numerous disjunctions to account for the potential divergences.

HACL*, now part of the EverCrypt project [19], is another cryptography library containing formally verified assembly code for x86-64 and PowerPC. The verification has been performed using the F* system. The generic modular inversion algorithm it provides is quite naive though, as it just performs a modular exponentiation. (Note that this approach was not considered to be naive at the time.) Regarding modular exponentiation, another formally verified implementation can be found in the WhyMP library [13]. Since timing attacks were not a concern for WhyMP, the code for modular exponentiation features all the subtle optimizations that are found in the state-of-the-art code of the GNU Multi Precision library. The code has been verified using the WHY3 platform and then extracted to C. Either way, algorithms based on modular exponentiation are quite different, and conceptually much simpler than those of Bernstein and Yang, Harrison, or Pornin.

We initially planned to achieve the whole proof of the algorithm using only WHY3, inspired by the success of the WhyMP project. Unfortunately, a large amount of time was spent fighting against the SMT solvers, trying to massage the verification conditions so that they would successfully be checked. At some point, it became apparent that, for most of the goals, it would be faster to prove them manually using the ROCQ proof assistant rather than to obtain them from the automated solvers.

Bernstein's mechanization in HOL Light has been instrumental in getting other people to formally verify the correctness of their low-level implementation. Our goal is similar, to help people verify their own variant of their own implementation of Pornin's algorithm. In that direction, a promising perspective is to conduct a full semi-interactive proof of functional correctness directly in EASYCRYPT [2], whose seamless integration of interactive reasoning with automated SMT discharge is well-suited to the non-linear and modular arithmetic obligations that arise in this algorithm, and would avoid the impedance mismatch we encountered between WHY3 and ROCQ. As a first step in this direction, we have conducted preliminary experiments in EASYCRYPT, verifying a convergence-driven variant of Pornin's algorithm, which suggest that this approach is viable. This setting also opens the door to verifying an actual high-performance implementation. The algorithm can be written in Jasmin, whose type-checker establishes constant-timeness of the generated assembly. The Jasmin program can then be extracted to EASYCRYPT, where memory safety and

³ s2n-bignum/x86/proofs/bignum_modinv.ml

functional correctness are discharged within the same semi-interactive framework. This is the methodology recently used by Almeida et al. [1] to deliver a machine-checked proof of IND-CCA security and functional correctness of ML-KEM, connecting a high-speed Jasmin implementation to its specification through EASYCRYPT. The same pipeline applies, on a smaller scale, to a verified implementation of Pornin’s algorithm. We expect the invariants and lemmas developed in this work to transfer to such an effort with limited adaptation, although more work is needed to reduce the impedance between the various tools involved.

References

- 1 José Bacelar Almeida, Santiago Arranz-Olmos, Manuel Barbosa, Gilles Barthe, François Dupressoir, Benjamin Grégoire, Vincent Laporte, Jean-Christophe Léchenet, Cameron Low, Tiago Oliveira, Hugo Pacheco, Miguel Quaresma, Peter Schwabe, and Pierre-Yves Strub. Formally verifying Kyber – Episode V: machine-checked IND-CCA security and correctness of ML-KEM in EasyCrypt. In Leonid Reyzin and Douglas Stebila, editors, *44th Annual International Cryptology Conference*, volume 14921 of *Lecture Notes in Computer Science*, pages 384–421, 2024. doi:10.1007/978-3-031-68379-4_12.
- 2 Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. EasyCrypt: A tutorial. In Alessandro Aldini, Javier Lopez, and Fabio Martinelli, editors, *Foundations of Security Analysis and Design VII*, volume 8604 of *Lecture Notes in Computer Science*, pages 146–166, 2014. doi:10.1007/978-3-319-10082-1_6.
- 3 Daniel J. Bernstein. Hull light, 2023. URL: <https://cr.yp.to/2023/hull-light-20230416.sage>.
- 4 Daniel J. Bernstein and Bo-Yin Yang. Fast constant-time gcd computation and modular inversion. Cryptology ePrint Archive, Paper 2019/266, 2019. URL: <https://eprint.iacr.org/2019/266>.
- 5 François Bobot, Jean-Christophe Filliâtre, Claude Marché, Guillaume Melquiond, and Andrei Paskevich. Preserving user proofs across specification changes. In Ernie Cohen and Andrey Rybalchenko, editors, *5th International Conference on Verified Software: Theories, Tools, and Experiments*, volume 8164 of *Lecture Notes in Computer Science*, pages 191–201, Menlo Park, CA, USA, May 2013. doi:10.1007/978-3-642-54108-7_10.
- 6 Richard P. Brent and Paul Zimmermann. *Modern Computer Arithmetic*. Cambridge Monographs on Applied and Computational Mathematics. Cambridge University Press, 2010.
- 7 Billy Bob Brumley and Nicola Taveri. Remote timing attacks are still practical. In Vijay Atluri and Claudia Díaz, editors, *16th European Symposium on Research in Computer Security*, volume 6879 of *Lecture Notes in Computer Science*, pages 355–371, September 2011. doi:10.1007/978-3-642-23822-2_20.
- 8 Sunjay Cauligi, Craig Disselkoen, Klaus von Gleissenthall, Dean M. Tullsen, Deian Stefan, Tamara Rezk, and Gilles Barthe. Constant-time foundations for the new Spectre era. In Alastair F. Donaldson and Emina Torlak, editors, *41st ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 913–926, June 2020. doi:10.1145/3385412.3385970.
- 9 John Harrison. The s2n-bignum library of formally verified ECC software. In *25th Workshop on Elliptic Curve Cryptography*, October 2024. URL: https://troll.iis.sinica.edu.tw/ecc24/slides/3-05-ecc2024_harrison.pdf.
- 10 Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In Neal Koblitz, editor, *16th Annual International Cryptology Conference*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113, 1996. doi:10.1007/3-540-68697-5_9.
- 11 Robert Y. Lewis and Paul-Nicolas Madelaine. Simplifying casts and coercions (extended abstract). In Pascal Fontaine, Konstantin Korovin, Ilias S. Kotsireas, Philipp Rümmer, and Sophie Tournet, editors, *7th Workshop on Practical Aspects of Automated Reasoning*, volume 2752 of *CEUR Workshop Proceedings*, pages 53–62, 2020. URL: <https://ceur-ws.org/Vol-2752/paper4.pdf>.

- 12 Assia Mahboubi and Enrico Tassi. *Mathematical Components*. Zenodo, October 2020. doi:10.5281/zenodo.3999478.
- 13 Guillaume Melquiond and Raphaël Rieu-Helfft. WhyMP, a formally verified arbitrary-precision integer library. *Journal of Symbolic Computation*, 115:74–95, 2023. doi:10.1016/j.jsc.2022.07.007.
- 14 Russell O'Connor and Andrew Poelstra. Formal verification of the Safegcd implementation, 2025. doi:10.48550/arXiv.2507.17956.
- 15 Christine Paulin-Mohring. Inductive definitions in the system Coq: rules and properties. In Marc Bezem and Jan Friso Groote, editors, *International Conference on Typed Lambda Calculi and Applications*, volume 664 of *Lecture Notes in Computer Science*, pages 328–345, March 1993. doi:10.1007/BFb0037116.
- 16 Thomas Pornin. Optimized binary GCD for modular inversion. Cryptology ePrint Archive, Paper 2020/972, 2020. URL: <https://eprint.iacr.org/2020/972>.
- 17 Kazuhiko Sakaguchi. Reflexive tactics for algebra, revisited. In June Andronick and Leonardo de Moura, editors, *13th International Conference on Interactive Theorem Proving*, volume 237 of *LIPICs*, pages 29:1–29:22, August 2022. doi:10.4230/LIPICs.ITP.2022.29.
- 18 Samuel Weiser, David Schrammel, Lukas Bodner, and Raphael Spreitzer. Big numbers – Big troubles: Systematically analyzing nonce leakage in (EC)DSA implementations. In Srdjan Capkun and Franziska Roesner, editors, *29th USENIX Security Symposium*, pages 1767–1784, August 2020. URL: <https://www.usenix.org/conference/usenixsecurity20/presentation/weiser>.
- 19 Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. HACl*: A verified modern cryptographic library. In David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM SIGSAC Conference on Computer and Communications Security*, pages 1789–1806, October 2017. doi:10.1145/3133956.3134043.