GCD: Garbled, Corrected, Demonstrandum

Fixing and Proving Go's Extended GCD Implementation

Linard Arquint
National University of Singapore
Singapore

Abstract

We verify the `extendedGCD` implementation in Go's standard library (`crypto/internal/fips140/bigmod`), which plays a crucial role in the generation of RSA key pairs. Even though the Go implementation is supposedly a direct port from BoringSSL's implementation, we uncovered two deviations that each break the algorithm's invariants: (1) the Go implementation deviates in the way coefficients are updated, and (2) it permits a larger input domain. We address both deviations; the first by fixing the Go implementation, which results in an on average 24 % speedup, and the second deviation by porting an existing proof for BoringSSL and extending it to cover the larger input domain. We prove correctness and termination of the fixed Go implementation using Gobra, a deductive program verifier for Go. Where necessary, we used Lean to prove key lemmata on non-linear arithmetic, which we import into Gobra.

Our verification effort reveals three key insights: subtle bugs can slip into even well-reviewed code with surprising ease; formal verification is a powerful tool for uncovering them; and AI agents can facilitate the verification process by iteratively refining invariants and lemmata based on Gobra's error messages.

Keywords

extended euclidean algorithm, go language, standard library, separation logic, automated verification, formal methods.

1 Introduction

Standard libraries are a critical part of the software ecosystem, as they provide fundamental building blocks for applications and are, thus, widely used and heavily scrutinized. The Go programming language provides numerous cryptographic functionality in its standard library, ranging from random number generation over RSA to TLS and SSH. Go's standard library implements algorithms approved by NIST's Federal Information Processing Standards (FIPS) 140-3 [19], which states security requirements for applications handling sensitive data in the US government and regulated industries such as finance and healthcare.

Correctness of a standard library and in particular its cryptographic code is crucial due to their widespread use and the security implications of bugs. Program verification enables us to rigorously reason about the correctness of an implementation, taking all possible inputs, execution paths, and memory states into account, thus uncovering bugs that may be missed by testing.

In this work, we focus on the implementation of the Extended Euclidean Algorithm, extended Greatest Common Divisor (GCD) for short, in Go's standard library. This implementation is crucial for RSA key generation and thus all applications that build atop RSA, including TLS, SSH, and PGP, as it computes the modular inverse of an RSA public exponent, which forms the core of any RSA private key. This modular inverse forms the core of any RSA private key, which is regulated by FIPS 140-3.

The currently used extended GCD implementation in Go's standard library was introduced in Go 1.24 in an effort to natively support FIPS 140-3 approved algorithms. Previous versions of Go relied on BoringSSL, Google's fork of OpenSSL, for FIPS-compliance, which however was limited to just a few supported target platforms.

Our results show that the currently used extended GCD implementation in Go's standard library differs from BoringSSL's implementation. These deviations are critical as the implementation claims to be a direct port of BoringSSL's implementation and claims correctness by referencing a proof for BoringSSL's implementation in the Rocq proof assistant [10, 21]. Despite code reviews by three reviewers, these deviations were not caught, which demonstrates how easily subtle bugs can be introduced even in well-reviewed code.

Contributions. We identify deviations of Go's extended GCD implementation w.r.t. to BoringSSL's implementation, propose a fix that achieves an on average speedup of 24 %, and prove its correctness and termination using Gobra [29], a deductive program verifier for Go. We build atop of BoringSSL's correctness proof, which we port from Rocq to Gobra and generalize to cover the entire input space that the Go implementation permits. By performing the proof directly on the Go implementation, we close the gap between the implementation and its proof, which allowed the discovered deviations to stay uncovered until now. Since Gobra takes only about 17 s to prove the implementation's correctness, we integrate the proof into continuous integration, which automatically checks the correctness of the implementation on every change. This integration ensures that we detect future deviations, which is not the case for BoringSSL's proof.

Disclosure. We disclosed the discovered deviations and proposed fixes to one of the primary code owners of the Go cryptography standard library on March 5, 2026. Together, we confirmed that the deviations result in at most an availability issue for RSA key generation. In particular, an incorrectly computed modular inverse is caught by defensive checks in the RSA key generation code, which would then return an error instead of an incorrect RSA key. The proposed fixes have passed code review and are scheduled for inclusion in Go 1.28 [3].

2 Extended GCD

The extended GCD algorithm computes, given integers a and n, the greatest common divisor $\gcd(a, n)$ together with the Bézout coefficients A and B satisfying $\gcd(a, n) = A * a - B * n$. When $\gcd(a, n) = 1$, the coefficient A is the modular inverse of a modulo

FCS '26, Lisbon, Portugal

n, i.e., $A * a \equiv 1 \pmod{n}$. This operation is fundamental in cryptography for RSA key generation as well as elliptic curve cryptography, which relies on modular inverses in prime fields.

RSA key generation uses the extended GCD algorithm in the following way. First, two large, secret prime numbers p and q are generated, and the public modulus is computed as $N = p * q$. Next, the key generation computes the secret exponent of the multiplicative group of integers modulo N, which is $\lambda(N) = \text{lcm}(p-1, q-1) = \frac{(p-1)*(q-1)}{\gcd(p-1,q-1)}$. Then, a public exponent e is chosen such that $1 < e < \lambda(N)$ and $\gcd(e, \lambda(N)) = 1$. Finally, the private key exponent d is computed as the modular inverse of e modulo $\lambda(N)$, i.e., $d \equiv e^{-1} \pmod{\lambda(N)}$. Thanks to the extended GCD algorithm, we can compute d efficiently because d corresponds to the Bézout coefficient A for e and $\lambda(N)$.

Extended GCD in Fiat Cryptography. Fiat Cryptography [14] is a framework for the Rocq proof assistant that enables expressing and proving cryptographic arithmetic in a high-level language and then extracts efficient, architecture-specific, and formally verified C implementations. While Fiat Cryptography has been used to prove the correctness of extended GCD as implemented in BoringSSL [10, 21], the actual implementation used in BoringSSL is handwritten, i.e., not extracted by Fiat Cryptography. Hence, there is no formal link between BoringSSL's implementation and its proof such that regressions in the implementation may go unnoticed unless the proof is updated accordingly.

The extended GCD algorithm used by BoringSSL loosely follows Algorithm 14.61 from the Handbook of Applied Cryptography [17], but merges loops, rearranges steps, and uses constant-time operations [10, 21]. We provide this algorithm in Alg. 1 but rewrite constant-time operations into variable-time operations for simplicity. Although the algorithm computes and stores the GCD in variable u, it does not return this value; instead, the algorithm returns the modular inverse if the GCD is 1 and an error otherwise. In each loop iteration, the algorithm halves either u or v and maintains the following invariants.

$$u = A * a - B * n \tag{1}$$

$$v = D * n - C * a \tag{2}$$

$$\gcd(u, v) = \gcd(a, n) \tag{3}$$

$$(\text{isodd}(u) \vee \text{isodd}(v)) \wedge (\text{isodd}(a) \vee \text{isodd}(n)) \tag{4}$$

with bounds $0 < n < 2^{\text{nbits}}$, $0 < a < 2^{\text{abits}}$, $a < n$, $0 < u \leq a$, $0 \leq v \leq n$, $0 \leq A < n$, $0 \leq B < a$, $0 \leq C < n$, $0 \leq D \leq a$. $\text{isodd}(x)$ denotes whether x is odd, i.e., $x \equiv 1 \pmod{2}$.

Each loop iteration consists of two parts: (1) if both u and v are odd, the algorithm subtracts the smaller from the larger, which ensures that exactly one of them becomes even, and updates the coefficients A, B, C, and D accordingly (lines 4–20); (2) halve either u or v (depending on which is even) and update the coefficients accordingly to maintain the invariants (lines 21–35). In-between these two parts, i.e., on line 21, exactly one of u and v is even and the invariants hold.

Extended GCD in Go. The Go standard library implements the extended GCD algorithm in the function `extendedGCD` in the `crypto/internal/fips140/bigmod` package. As stated in the description of the corresponding change [27], the Go implementation is supposedly a direct port of BoringSSL's implementation, and code comments point to Fiat Cryptography's proof as justification for the Go implementation's correctness.

The `extendedGCD` function returns not only the Bézout coefficient A (like BoringSSL) but also the GCD value itself. While `extendedGCD` is a package-private function, the standard library exposes these return values through two different functions to clients.

Algorithm 1 Binary Extended GCD in BoringSSL and Fiat Cryptography. For simplicity, we rewrite constant-time into variable-time operations.

Require: Two positive integers a and n, such that $a \neq 0$, $n \neq 0$, $a < n$, and not both a and n are even. abits and nbits are the bit lengths of a and n, respectively.
Ensure: Error or integer A, such that $(A * a) \equiv 1 \pmod{n}$.

```
1:  u ← a,   v ← n,   i ← 0
2:  A ← 1,   B ← 0,   C ← 0,   D ← 1
3:  while i < abits + nbits do
    invariant (1) ∧ (2) ∧ (3) ∧ (4)
4:      if u and v are both odd then
5:          if v < u then
6:              u ← u − v
7:              if A + C < n then
8:                  A ← A + C,   B ← B + D
9:              else
10:                 A ← A + C − n,   B ← B + D − a
11:             end if
12:         else
13:             v ← v − u
14:             if A + C < n then
15:                 C ← A + C,   D ← B + D
16:             else
17:                 C ← A + C − n,   D ← B + D − a
18:             end if
19:         end if
20:     end if
21:     if u is even then
22:         u ← u/2
23:         if A or B is odd then
24:             A ← (A + n)/2,   B ← (B + a)/2
25:         else
26:             A ← A/2,   B ← B/2
27:         end if
28:     else
29:         v ← v/2
30:         if C or D is odd then
31:             C ← (C + n)/2,   D ← (D + a)/2
32:         else
33:             C ← C/2,   D ← D/2
34:         end if
35:     end if
36:     i ← i + 1
37: end while
38: return if u = 1 then A else error end if
```

`GCDVarTime` returns the GCD value, while `InverseVarTime` returns the modular inverse if the GCD is 1, and an error otherwise.

The implementation's correctness is crucial as RSA key generation within Go's standard library uses both functions. `GCDVarTime` is used to compute $\lambda(N)$, while `InverseVarTime` is used to compute the private exponent d as the modular inverse of the public exponent e modulo $\lambda(N)$.

Correctness of the implementation is claimed [27] only by code comments pointing to the Fiat Cryptography proof. In particular, a comment lists the very same algorithmic changes as also found in the Fiat Cryptography proof documenting BoringSSL's changes w.r.t. the original Algorithm 14.61 from the Handbook of Applied Cryptography. No comment indicates that the Go implementation performs any additional algorithmic changes. However, we found that the Go implementation deviates substantially from the BoringSSL and Fiat Cryptography implementation, which is critical as the code comments imply that the correctness of the Go implementation directly follows from Fiat Cryptography's proof.

We found two classes of deviations in the Go implementation w.r.t. the BoringSSL and Fiat Cryptography implementation. The first class consists of two major but subtle deviations; major as each such deviation breaks the algorithm's invariants, and subtle as neither the code's author nor three code reviewers spotted them. We found these two deviations only after a failed proof attempt and discuss them in detail in Sec. 3.

The second class consists of three minor deviations. First, the Go implementation uses variable-time operations and, thus, features more complex control flow, making a statement-by-statement correspondence between the two implementations non-trivial. Second, the Go implementation loops until v becomes zero, while BoringSSL's and Fiat Cryptography's implementation uses a fixed number of iterations, which corresponds to the sum of the bit lengths of the inputs. This impacts the termination argument. Third, the Go implementation of `extendedGCD` does not return an error when the GCD value is different from 1, while BoringSSL does. We prove correctness of these minor deviations by adapting Fiat Cryptography's proof in a straightforward way. We report on the full proof effort in Sec. 4.

3 Major Deviations

We report and address the two major deviations that we found while carrying out the proofs on the Go implementation. While they are subtle and, thus, have slipped through three code reviews, both deviations are critical as each one breaks the algorithm's invariants.

3.1 Unsynchronized Subtractions Deviation

The first major deviation that we found in the Go implementation concerns the updating of the coefficients A, B, C, and D in the first part of every loop iteration, namely lines 7–11 and lines 14–18 in Alg. 1. BoringSSL and Fiat Cryptography consider whether the sum $A + C$ exceeds the modulus n to decide whether to reduce $A + C$ *and* $B + D$ by subtracting n and a, respectively, thus, ensuring that both reductions happen synchronously. Synchronizing these reductions is crucial for maintaining the loop invariants (1) and (2). We illustrate this by focusing on the case where u and v are both odd and $v < u$ (the case where $v \geq u$ is analogous). At the beginning

```
1  carry := A.add(C)
2  B.add(D)
3  if choice(carry) == yes || A.cmpGeq(m) == yes {
4    A.sub(m)
5    B.sub(a)
6  }
```

Figure 1: Proposed synchronized reduction for updating the coefficients A and B. Analogously, we propose the same fix for updating the coefficients C and D.

of a loop iteration, the invariants hold, and the updates to u, A, and B maintain the invariants as follows. We use primed variables to denote the updated values. Ignore the summands highlighted in blue if $A + C < n$. If $A + C \geq n$, the algorithm performs a synchronized reduction using the highlighted summands, which cancel out.

$$\begin{aligned} u' &= u - v \\ &= (A * a - B * n) - (D * n - C * a) \\ &= A * a - B * n - D * n + C * a \boxed{- n * a + n * a} \\ &= (A + C \boxed{- n}) * a - (B + D \boxed{- a}) * n \\ &= A' * a - B' * n \end{aligned}$$

The Go implementation, however, performs unsynchronized reductions by updating the coefficients as

$A \leftarrow$ **if** $A + C < n$ **then** $A + C$ **else** $A + C - n$ **end if**
$B \leftarrow$ **if** $B + D < a$ **then** $B + D$ **else** $B + D - a$ **end if**

and

$C \leftarrow$ **if** $A + C < n$ **then** $A + C$ **else** $A + C - n$ **end if**
$D \leftarrow$ **if** $B + D < a$ **then** $B + D$ **else** $B + D - a$ **end if**

instead of lines 7–11 and lines 14–18, respectively. This breaks the invariants (1) and (2) as the reductions leave one of the two summands highlighted in blue without a corresponding canceling summand. Since we have not found any input yet that causes an incorrect output, it is likely that different invariants could establish the original implementation's correctness. We have not pursued this direction further and instead propose a fix that resolves this deviation, which the code's author confirmed to be unintentional, as well as improves runtime performance (cf. Sec. 4.4).

Fix. We propose and implemented a fix that synchronizes the reductions by using the same condition for both pairs of coefficients. Fig. 1 shows the proposed fix for updating the coefficients A and B. `x.add(y)` (respectively `x.sub(y)`) compute $x \mathrel{+}= y$ ($x \mathrel{-}= y$) on the arbitrary-length integer representation and return the carry (borrow) flag. The branch condition checks whether the carry flag is set or $A \geq m$. We prove (cf. Sec. 4) that updating A and B in this way is mathematically equivalent to lines 7–11 in Alg. 1. This shows that our fix aligns Go's coefficient updates with those of BoringSSL and Fiat Cryptography at the algorithmic level, even though the latter two implement these updates using constant-time operations.

3.2 Input Domain Deviation

As the second major deviation, we found that the Go implementation permits a larger input domain than BoringSSL and the proven Fiat Cryptography implementation. This is critical as this larger input domain breaks the invariants on which the proof relies.

Alg. 1 specifies as a precondition that $a < n$ must hold[1]. The Go implementation, however, does not specify this requirement for its `GCDVarTime` function, which returns the GCD value. Dropping this requirement is natural as the mathematical definition of GCD is symmetric in its two arguments, and thus, does not require any ordering assumption.

Addressing this deviation requires proof changes only, not code changes. We ran targeted tests to gain confidence that `GCDVarTime` produces correct results even when $a \geq n$, and we extended our formal proof to also cover the case $a \geq n$ by relaxing the invariants in the Go implementation. Specifically, the invariants (1), (2), $a < n$, and the bounds for the coefficients A, B, C, and D hold only when $a < n$. In addition, we adapted the postcondition of `extendedGCD` to reflect the different guarantees that hold in the two cases $a < n$ and $a \geq n$. Only the former case guarantees that the coefficient A is the modular inverse of a modulo n, while both cases guarantee that $u = \gcd(a, n)$ holds.

4 Verified Go Implementation

In this section, we describe our verified extended GCD implementation in Go. All code, proofs, and continuous integration scripts are available open-source [4]. First, we introduce the Gobra program verifier and illustrate necessary background on excerpts from our verified implementation in Sec. 4.1. Sec. 4.2 presents the specifications capturing correctness and termination, against which we verify the implementation using Gobra. We cover our assumptions in Sec. 4.3, evaluate our proposed code changes and verification effort in Sec. 4.4, and discuss our results and the use of AI agents in Sec. 4.5.

4.1 Gobra

Gobra [29] is a deductive, separation-logic-based program verifier for Go, which has been successfully applied to verifying parts of the WireGuard VPN protocol implementation in Go [7, 8], an Internet router for the SCION Internet architecture [20], and a fork of the AWS SSM Agent [6]. It translates annotated Go programs into the Viper intermediate verification language [18], which uses symbolic execution and discharges proof obligations via the Z3 SMT solver.

Modular verification. Gobra performs *modular* verification: each function is verified in isolation against its specification, which consists of preconditions (`requires`) and postconditions (`ensures`). A function verifies successfully if executing the function body in an arbitrary state satisfying the precondition is guaranteed to result in a state satisfying the postcondition. A caller must establish the preconditions of its callee and can safely assume the callee's postconditions. This decomposition allows proofs to scale, since the verification of each function depends only on the specifications of its callees, not their implementations.

Permissions. Gobra uses separation logic [23] to reason about heap-allocated objects, side effects, and concurrency. Every heap access requires that the current thread possesses the *permission* for that heap object. A permission for the heap object pointed to by pointer `ptr` is expressed as `acc(ptr, p)`, where `p` is a fractional [12] in the interval $(0, 1]$. A full permission ($p = 1$) grants both read *and* write access, while any fraction strictly greater than zero and smaller than 1 grants read-only access. Permission for a particular heap object can be split and combined, e.g., to share read access among multiple threads. Since permissions are non-duplicable, a proof in separation logic guarantees memory safety and absence of data races: if two threads have sufficient permissions to access the same heap object, they both must have a non-zero permission for this heap object, which is possible as long as the sum of these permissions does not exceed 1. This ensures that these threads perform only read but no write accesses, unless the threads synchronize their accesses by using a suitable concurrency primitive[2].

```
1  type Nat struct {
2    limbs []uint
3  }
4  /*@
5  pred (n *Nat) Inv() {
6    acc(n) &&
7    (forall j int :: 0 ≤ j < len(n.limbs) ==>
8      acc(&n.limbs[j])) &&
9    let allLimbs := n.limbs[:cap(n.limbs)] in
10   (forall j int :: len(n.limbs) ≤ j < len(allLimbs) ==>
11     acc(&allLimbs[j]) && allLimbs[j] == 0)
12 }
13 @*/
```

Figure 2: Definition of the `Nat` struct that the Go standard library uses to represent arbitrary-length natural numbers and our invariant predicate.

Predicates. Permissions can be grouped into *predicates*, such as the `Nat.Inv()` predicate (cf. Fig. 2) used throughout our verification. `Nat` is a struct type representing arbitrary-length natural numbers. Internally, this type uses a slice of machine words, called *limbs*, to represent the number in a little-endian manner. A slice in Go provides a dynamic view into an underlying fixed-size array. The `Nat.Inv()` predicate bundles the permissions for a `Nat`'s struct field (line 6), all its limbs up to the slice's length (lines 7–8), and the limbs between the slice's length and capacity (lines 9–11)[3]. In addition, lines 9–11 state the invariant that limbs beyond the current length are zero, which was crucial for the correctness of the `Nat.reset` function up until recently (cf. Sec. 4.5).

Callers pass permissions to `Nat.Inv()` in preconditions, e.g., `n.Inv()` to grant the callee read and write access to `n`. Passing a p fraction of this invariant to a callee (`acc(n.Inv(), p)`) conceptually passes p fractions of all permissions bundled in `Nat.Inv()`. Predicates are opaque, meaning that accessing the permissions bundled in a predicate instance requires unfolding the predicate first, which exchanges the predicate instance for its definition. To simplify the presentation, we use full permissions in the remainder of this paper. However, we use more permissive specifications for our verified functions in the codebase, which distinguish between read-only and write access for inputs.

Ghost code. Gobra supports *ghost code*: annotations that exist solely for verification and do not alter a program's runtime behavior. Ghost code is delimited by `//@` (single-line) or `/*@ ... @*/`

[1] Both BoringSSL and Fiat Cryptography state this precondition. In addition, BoringSSL checks this condition at runtime and returns an error if unmet.

[2] Proof-theoretically, a concurrency primitive can be used to allow threads to exchange permissions with each other, and, thus, obtain full permissions within a mutually-exclusive, critical section, which justifies certain write accesses therein.

[3] We split the permissions to the elements of the underlying array into two separate forall quantified assertions for automation reasons.

```
/*@
ghost
opaque
requires n.Inv()
ensures  0 <= res && res < n.ValCount()
decreases
pure func (n *Nat) Repr() (res integer) {
  return ...
}

ghost
opaque
requires n.Inv()
ensures  0 <= res
decreases
pure func (n *Nat) AnnLen() (res int) {
  return unfolding n.Inv() in len(n.limbs)
}
@*/
```

Figure 3: Ghost functions providing an abstraction for `Nat` by mapping it to its unbounded, numeric value and exposing its limbs' length. `Nat.Repr`'s postcondition states that `res` $\in [0, 2^{\texttt{bits.UintSize}*\texttt{len(n.limbs)}})$.

```
// extendedGCD computes u and A such that
// u = GCD(a, n) = A*a - B*n. u will have the size of
// the larger of a and n, and A will have the size of
// n. It is an error if either a or n is zero, or if
// they are both even.
//@ requires a.Inv() && n.Inv()
//@ ensures  a.Inv() && n.Inv()
//@ ensures  err == nil ==> u.Inv() && A.Inv()
//@ ensures  err == nil ==>
//@   u.Repr() == gcd(a.Repr(), n.Repr())
//@ ensures  err == nil && a.Repr() < n.Repr() ==>
//@   u.Repr() == A.Repr()*a.Repr() - BRepr*n.Repr()
//@ ensures  err == nil ==>
//@   u.AnnLen() == gmax(a.AnnLen(), n.AnnLen())
//@ ensures  err == nil ==> A.AnnLen() == n.AnnLen()
//@ decreases
func extendedGCD(a, n *Nat
) (u, A *Nat, err error /*@, ghost BRepr uint @*/)
```

Figure 4: Specification for the verified `extendedGCD` function. For presentation purposes, we use full instead of fractional permissions for `a` and `n` and omit that `a` and `n` remain unmodified. `gmax` returns the maximum of its arguments.

(multi-line) comments, making it invisible to the Go compiler. Ghost annotations include ghost parameters, ghost variables, and ghost functions (marked ghost). For instance, `Nat.Repr()` in Fig. 3 is a pure function that returns the numeric value represented by a `Nat`'s limbs. `pure` indicates that a function is free of side effects, which allows us to call it in specifications and loop invariants.

Since the length of the limbs is crucial, as manifested by numerous natural language specifications referring to a `Nat`'s "announced length", we additionally define a ghost function `Nat.AnnLen()`, which allows us to easily refer to the limbs' length in specifications and proofs. To optimize verification time, we mark some functions as `opaque`, hiding their implementation by default. Their implementation is revealed at specific proof points only, via a `reveal` operation, to connect a `Nat`'s abstraction to its implementation.

Ghost parameters are useful for avoiding existential quantification in specifications. For example, `extendedGCD` (cf. Sec. 4.2) returns the ghost value `BRepr` alongside its regular results, allowing the postcondition to state the Bézout identity `u.Repr() == A.Repr() * a.Repr() - BRepr * m.Repr()` without having to existentially quantify `BRepr`.

The `trusted` keyword. Functions annotated with `trusted` are assumed to satisfy their specification without verification. We use `trusted` for bit-level operations on `Nat`, whose correctness depends on low-level arithmetic properties that Gobra cannot yet verify (e.g., `cmpGeq`, `add`, `sub`, and `rshift1`). In contrast, all our lemmata are either verified by Gobra or backed by Lean proofs included as comments in the source code. The latter proved useful for non-linear arithmetic lemmata that Z3 could not handle.

To focus our verification efforts on `extendedGCD` and its clients, we marked several non-ghost functions in the `crypto/internal/fips140/bigmod` package as `trusted`. We equip them with a trivially unsatisfiable precondition (i.e., `requires false`) to make it explicit that these functions are irrelevant for `extendedGCD` *and* their behavior is not yet faithfully captured in their specification. Since this precondition can never be established, we ensure that these functions are not called from any verified code, preventing unsound assumptions about their behavior from creeping into the proof of `extendedGCD`.

4.2 Specifications and Proofs

Thanks to the code and proof fixes described in Sec. 3, we successfully verify the extended GCD implementation in Go. In this subsection, we present the specifications of the internal `extendedGCD` function and the two functions `InverseVarTime` and `GCDVarTime`, which are the two clients of `extendedGCD`. We end with covering key lemmata that we use throughout the proof.

`extendedGCD`. Fig. 4 shows the specification of `extendedGCD` against which we successfully verify the implementation using Gobra. Our specification is a direct formalization of the natural language specification in the standard library, which is included at the top of the code listing. Due to our finding in Sec. 3.2, lines 11–12 state that $u = A * a - B * n$ holds only if $a < n$. Additionally, `decreases` instructs Gobra to also prove that the function terminates.

Within the implementation of `extendedGCD`, we prove its loop using the adapted loop invariants as described in Sec. 3.2. For automation reasons, we hide the invariants (1) and (2) (that hold in the case that $a < n$), in opaque pure functions and reveal their definitions only where necessary. This does not impact soundness but speeds up verification significantly by avoiding exposing non-linear arithmetic to the SMT solver for most of the proof. We use `u.Repr() + v.Repr()` as a termination measure for the loop, which instructs Gobra to prove that this expression strictly decreases in each iteration and is bounded from below, which implies that the loop terminates.

`InverseVarTime` and `GCDVarTime`. `InverseVarTime` computes the modular inverse of `a` modulo `n` and `GCDVarTime` computes the GCD of `a` and `b`. Both functions internally invoke the `extendedGCD` function, and we prove them correct against the specifications in Fig. 5. Both functions store the result in `x`, which is also returned as the first return parameter `r`.

```
1  // InverseVarTime calculates x = a^-1 mod n and returns (x, true) if a is invertible. Otherwise, InverseVarTime
2  // returns (x, false) and x is not modified. a must be reduced modulo n, but doesn't need to have the same size. The
3  // output will be resized to the size of n and overwritten.
4  //@ requires x.Inv() && a.Inv() && n.Inv()
5  //@ requires a.Repr() < n.Repr()
6  //@ ensures  x.Inv() && a.Inv() && n.Inv()
7  //@ ensures  r == x
8  //@ ensures  ok ==> gcd(a.Repr(), n.Repr()) == 1
9  //@ ensures  ok ==> gcd(a.Repr(), n.Repr()) == x.Repr() * a.Repr() - BRepr * n.Repr()
10 //@ ensures  ok ==> x.AnnLen() == n.AnnLen()
11 //@ ensures !ok ==> x.Repr() == old(x.Repr()) && x.AnnLen() == old(x.AnnLen())
12 //@ decreases
13 func (x *Nat) InverseVarTime(a *Nat, n *Modulus) (r *Nat, ok bool /*@, ghost BRepr uint @*/)

15 // GCDVarTime calculates x = GCD(a, b) where at least one of a or b is odd, and both are non-zero. If GCDVarTime
16 // returns an error, x is not modified. The output will be resized to the size of the larger of a and b.
17 //@ requires x.Inv() && a.Inv() && b.Inv()
18 //@ requires a.Repr() % 2 == 1 || b.Repr() % 2 == 1
19 //@ requires a.Repr() != 0 && b.Repr() != 0
20 //@ ensures  x.Inv() && a.Inv() && b.Inv()
21 //@ ensures  err == nil ==> r == x
22 //@ ensures  err == nil ==> x.Repr() == gcd(a.Repr(), b.Repr())
23 //@ ensures  err == nil ==> x.AnnLen() == gmax(a.AnnLen(), b.AnnLen())
24 //@ ensures  err != nil ==> x.Repr() == old(x.Repr()) && x.AnnLen() == old(x.AnnLen())
25 //@ decreases
26 func (x *Nat) GCDVarTime(a, b *Nat) (r *Nat, err error)
```

Figure 5: Specifications for the verified `InverseVarTime` and `GCDVarTime` functions. We make similar presentation choices as in Fig. 4, and `Modulus.Inv()` is a predicate similar to `Nat.Inv()` that bundles permissions for a `Modulus` struct. `old(e)` refers to the value of expression `e` in a function's pre-state, allowing us to express that a `Nat` is not modified in case of failure.

Directly following the standard library's natural language specifications, line 5 expresses that `a` must be reduced modulo `n`. Similarly, line 11 states that `x` is not modified on failure. `GCDVarTime` follows the same pattern, where line 18 expresses that at least one of `a` or `b` must be odd, while the next line states that both `a` and `b` must be non-zero.

Key Lemmata. We define and prove eight lemmata, which are crucial for the proof of `extendedGCD` and most of which involve non-linear integer arithmetic. Gobra verifies four of them, while we provide machine-checked Lean proofs for the other four lemmata. For the latter four lemmata, we include the Lean proof as comment in the codebase while marking the corresponding lemma as `trusted` for Gobra. Two of these four Lean-checked lemmata have a direct counterpart in Fiat Cryptography, while the other two state basic properties of modular arithmetic that Z3 cannot derive on its own. Namely,

$$0 < b < a \implies a \bmod b = (a - b) \bmod b \qquad \text{and}$$
$$0 \le a \wedge 0 \le b \implies (a * b) \bmod 2 = (a \bmod 2) * (b \bmod 2).$$

4.3 Assumptions

Our verification rests on the following assumptions.

Gobra and Z3. We trust the correctness of the Gobra verifier and the underlying SMT solver. Since, Gobra translates Go programs to Viper, which encodes verification conditions in SMT, a bug in any of these components could lead to unsound verification. This risk is mitigated by choosing a widely-used SMT solver, namely Z3, and having substantial testing frameworks for both Gobra and Viper.

Trusted functions. We assume that all functions annotated with `trusted` (and without `requires false`) satisfy their specifications. These functions primarily perform bit-level operations. Verifying their implementation would require better support for bit-level reasoning in Gobra, which is a promising direction for future work.

While we assume that these functions satisfy their specifications, these functions are relatively straightforward and consist of at most 13 lines of code each, making it feasible to review them manually and gain confidence in their correctness. Similarly, our four trusted lemmata (cf. Sec. 4.2) are formalized in Lean. While the Lean proofs are machine-checked and the translation from Gobra to Lean is straightforward and has been manually reviewed, their remains a small trust gap as we switch from one formalism to another.

Overflows. Gobra currently treats `int` and `uint` as unbounded types. This means that our verification does not detect potential integer overflows in intermediate computations. In our setting, this is mitigated by the fact that the `Nat` type manages multi-precision arithmetic internally. In particular, the trusted functions are assumed to handle limb-level overflows and return carry and borrow flags accordingly. Given these trusted functions, we prove that the verified functions correctly consider the returned carry and borrow flags for their computations. Ongoing work in Gobra aims to improve the support for overflow checking, which would allow us to verify the absence of overflow bugs in the future.

4.4 Evaluation

We evaluate our work by measuring the performance impact of our proposed code change and the verification effort.

Performance Evaluation. Sec. 3.1 described a deviation of the Go implementation from the algorithm used in BoringSSL and Fiat Cryptography, which breaks the algorithm's invariants. We address this deviation with a fix (cf. Fig. 1), raising the question to what extent this fix impacts the implementation's runtime performance.

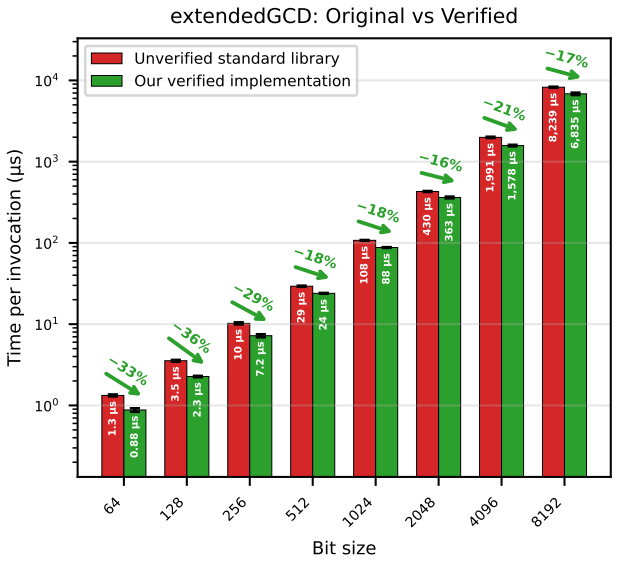

Figure 6: Performance comparison between the original and our verified implementations for different limb sizes. Lower is better. Each bar shows the arithmetic average runtime across 4 input pairs and 10 runs per input pair, with the average value labeled within each bar and error bars showing the standard deviation. The green arrows indicate the percentage speedup achieved by our implementation.

We answer this question by benchmarking the original and fixed implementations of `extendedGCD`, since our proposed fix is limited to this function. Its clients, `InverseVarTime` and `GCDVarTime`, remain unchanged, and we expect their performance to match `extendedGCD`'s, as both are thin wrappers that only handle errors and copy the result.

We use Go's built-in benchmarking framework to measure the runtime of `extendedGCD` for different limb sizes, namely 2^k for $k \in [0, 7]$. For this purpose, we generate 4 input pairs (a, n) per limb size, where each limb in a and n is initially randomly sampled, and we ensure that the following properties hold: n's most significant limb is non-zero (otherwise, we resample this limb), n is odd by setting the least significant bit to 1, and $0 < a < n$ by taking a modulo n and checking that the result is strictly positive (otherwise, we resample a and start over). For each input pair, we perform 10 runs for both the original and fixed implementations, where each run uses `testing.B.Loop` [15, 24], Go's preferred way of writing benchmarks. `testing.B.Loop` repeatedly invokes `extendedGCD` with the same input pair until the total runtime exceeds the threshold of 1 s to amortize measurement overhead.

Fig. 6 visualizes our results, showing the arithmetic average runtime and standard deviation for each limb size and implementation. In particular, the standard deviation includes both the variability across 4 input pairs and the variability across different runs for the same input pair. For all measured limb sizes, our fixed and verified implementation is at least 15.6 % faster than the original implementation. Across all benchmarks, we obtain a statistically significant, geometric mean performance improvement of 23.98 %.

The main difference between the original and fixed implementations beyond the synchronization of the coefficient updates (cf. Sec. 3.1) is the way intermediate results are stored (1) and computed (2). In terms of storage (1), each loop iteration that triggers the loop body's first part (lines 4–20 in Alg. 1), allocates in the original implementation two buffers for storing intermediate results if the intermediate results use up to 2048 bits, and four buffers otherwise. In the latter case, the number of allocations doubles because Go's `Nat` type allocates a 2048 bit buffer in its constructor, which ends up being too small for storing the intermediate results, leading to an additional buffer allocation per intermediate result. In contrast, our verified implementation performs in-place updates and, thus, allocates zero buffers in the loop body. In terms of computations (2), the original implementation performs, in addition to the operations given on lines 4–20, two zeroing operations for limbs sizes $\leq$ 2048 bits, two copying operations, and two conditional copy operations, each iterating over all limbs of a `Nat`, while saving one comparison operation. In contrast, our verified implementation performs only the operations as stated on lines 4–20 in Alg. 1.

Function	LOC	LOS
`extendedGCD`	53	99
`InverseVarTime`	10	18
`GCDVarTime`	7	12
`syncAdd`	8	60
Other Go functions	71	314
Auxiliary definitions	–	130
Lemmata	–	349
Total	149	838

Table 1: Overview over the verified lines of code (LOC) and lines of specification (LOS) (incl. ghost code) in Gobra.

Proof Evaluation. We successfully verify the fixed `extendedGCD` implementation and its callers `InverseVarTime` and `GCDVarTime` against their specifications using Gobra. In order to prove termination and their correctness, we require specifications, which include loop invariants, define and prove lemmata (cf. Sec. 4.2), and state and assume specifications for trusted functions (cf. Sec. 4.3). Here, we report on the proof size and the verification time.

Tab. 1 summarizes our verified code, measured in lines of code (LOC) and lines of specification (LOS), where the latter includes ghost code. A line is counted as a line of code *and* a line of specification if it contains both code and specification elements, which is, e.g., the case for a function signature that includes a ghost parameter, like the last line in Fig. 4. `syncAdd` implements the coefficient updates using synchronized reduction, whose implementation without any proof-related specifications is shown in Fig. 1. Auxiliary definitions and lemmata are exclusively ghost code. In particular, auxiliary definitions consist of predicates and pure functions (cf. Sec. 4.1), while lemmata derive, e.g., mathematical properties about GCD or non-linear arithmetic that are crucial for the proof of `extendedGCD`. While 22 lines of lemmata are trusted, they are backed by 36 lines of Lean proofs establishing their correctness. Since Tab. 1 reports

verified lines, the table does *not* include `trusted` functions that are neither verified by Gobra nor backed by Lean proofs, which include bit-level operations on `Nat`. We have 14 trusted functions like `Equal`, `IsZero`, `IsOdd`, and `add`, whose specifications and implementations are relatively straightforward and easy to review manually, amounting to a total of 122 LOC and 91 LOS.

All functions, auxiliary definitions, and lemmata together are verified in 16.9 s. We have measured the verification times by computing the 10 % Winsorized mean of the wall-clock runtime across 20 verification runs on a 2024 Apple MacBook Pro with M4 Pro processor, macOS Tahoe 26.3.1, Gobra v26.02, and Z3 4.8.7 arm64. This short verification time allows us to perform continuous verification, i.e., we integrated Gobra into a GitHub continuous integration workflow to verify the implementation after each code change [5].

4.5 Discussion

Modern program verifiers like Gobra are powerful tools that significantly increase the confidence in the correctness of complex implementations, such as the extended GCD algorithm in Go's standard library. Although Gobra would require better support for bit-level operations to close the remaining trust gap in our verification, we were able to successfully verify the algorithm's implementation building atop of a set of small, trusted functions. Effectively, we break down the correctness of the overall implementation, whose correctness argument goes beyond what we can realistically expect from code reviewers, into the correctness of a handful of trusted functions. We are convinced that these trusted functions are small and straightforward enough, such that reviewers can convince themselves of their correctness w.r.t. their specifications, which state the expected behavior in mathematical terms. This achieves a good balance between verification effort and the gained confidence in the overall implementation's correctness.

Besides providing a mean to break down the overall correctness argument, program verifiers impose a discipline of reasoning about correctness that helps uncover subtle issues. We initially worked on the go1.24.0 version of the codebase, and realized that the implementation's correctness relies on an implicit invariant about the `Nat` type, namely that all limbs beyond the announced length must be zero; otherwise, changing the announced length could lead to a change in the represented value. Our initial predicate definition was too weak to capture this invariant, which lead to the addition of `allLimbs[j] == 0` on line 11 in Fig. 2. While the Go developers independently discovered and addressed this brittleness [28] within about three months, the discipline of writing down invariants and reasoning about how the implementation relates to them led us to the same discovery in much less time.

Experience with Agentic Verification. We used Anthropic's AI agent Claude Code throughout the verification effort, which reduced the manual effort significantly. We estimate that we spent around two person-weeks on the verification, which includes prompting the agent. We note the following observations.

Applying the agent was effective because Gobra produces insightful error messages that the agent could interpret and act upon. This allowed the agent to autonomously iterate on both syntactic (e.g., missing ghost arguments) and semantic issues, where the latter includes too weak or too strong preconditions, postconditions, and loop invariants as well as missing lemma invocations. While the agent figured out on its own that moving some non-linear arithmetic reasoning into lemmata helps in avoiding timeouts in Z3, we taught the agent about `opaque pure` functions to further reduce the impact of non-linear arithmetic, after which the agent effectively used them for the loop invariants (1) and (2).

When the agent repeatedly failed to prove the invariants for the original coefficient update, it correctly suggested that the independent reduction of the coefficients might be the root cause. This hint pointed us in the right direction to identify the unsynchronized subtractions deviation (Sec. 3.1). However, the agent initially treated the implementation as authoritative—assuming the code was correct and looking for a matching proof—rather than questioning the implementation's correctness. We had to prompt the agent several times to compare the Go implementation with Fiat Cryptography's algorithm and proof before it eventually identified the deviation. Once identified, the agent was able to suggest a fix and successfully verify the fixed implementation.

While the agent eventually produced a valid proof, we performed several iterations of manual cleanup to combine, simplify, and rename the generated lemmata. Hence, the final proof structure that uses two invocations of an `opaque pure` function to express the invariants (1) and (2), and a few high-level lemmata is significantly cleaner than the intermediate versions that the agent produced. Understanding the implementation and the generated proof as well as performing this cleanup constitutes the majority of the two person-weeks we spent on the verification effort. Nevertheless, we estimate that the agent significantly reduced the verification effort by providing a valid proof that we could then clean up, rather than requiring us to develop the proof details from scratch.

5 Related Work

We compare our work to related efforts in the area of verified cryptographic implementations, with a focus on the extended GCD algorithm and modular inverse computations.

Fiat Cryptography [14] is closest to our work, as it also provides a verified extended GCD implementation. Fiat Cryptography is a framework for the Rocq proof assistant that allows to express and prove algorithms in a high-level language and then extract efficient C code. While BoringSSL has adopted Curve25519 and P-256 implementations generated by Fiat Cryptography and, thus, benefits from Fiat Cryptography's guarantees, the story for the extended GCD implementation is different. BoringSSL uses a handwritten C implementation of the extended GCD algorithm, which has been ported and proven correct in Fiat Cryptography. Since there is no formal link between BoringSSL's and Fiat Cryptography's extended GCD implementations, subtle differences or regressions may go unnoticed. In contrast, we reason about implementations on the code-level, which allows us to verify *existing* implementations and does not require implementers to exchange their handwritten implementations for generated ones.

EverCrypt [22] is a verified cryptographic library that bundles HACL⋆ [30] and Vale [11]. HACL⋆ is a library written and verified in F⋆ [25], which extracts to C, while Vale is a tool for writing low-level code in a C-like language that is verified in either Dafny [16] or F⋆ and extracts to assembly. While we are not aware of any GCD

or modular inverse implementations in Vale, HACL⋆ provides only limited support for computing modular inverses, which is insufficient for RSA key generation. In particular, HACL⋆ implements two different approaches to compute the modular inverse, neither of which is as general as the extended GCD algorithm (which we verify). `mod_inv_limb` computes the modular inverse but is limited to single-limb moduli (we support multi-limb moduli), while `bn_mod_inv_prime` computes the modular inverse for multi-limb numbers using Fermat's little theorem and, thus, requires a prime modulus (which we do not require and is not the case for RSA key generation). The same limitations apply to libcrux [13], a Rust cryptographic library, because the library provides Rust implementations of the same two functions by translating their HACL⋆ implementations to Rust.

The Formosa Crypto project [26] comes with a language and a provably semantics-preserving compiler called Jasmin [1] to write fast and secure low-level assembly code. To prove properties about Jasmin code, the compiler either checks certain properties itself, e.g., using a type-system to enforce constant-time programming, or extracts an EasyCrypt [9] model about which one can prove properties using the EasyCrypt proof assistant. While Jasmin's arbitrary-length number library libjbn [2] computes modular inverses using Fermat's little theorem (thus, the same limitations apply as for HACL⋆), we are not aware of any executable Jasmin implementation of the extended GCD algorithm.

6 Conclusions

We have applied Gobra to the `extendedGCD` implementation in Go's standard library (`crypto/internal/fips140/bigmod`), uncovering two deviations that each break the algorithm's invariants. Our suggested fixes resolve both deviations, while significantly improving the implementation's performance. Using Gobra, we prove the fixed implementation correct against its formal specification, which mirrors the natural language specification provided in the Go standard library. Extending Gobra with stronger support for bit-level reasoning would enable us to verify functions that are currently trusted, thereby reducing our trust assumptions and further increasing confidence in the implementation.

Acknowledgments

We thank Filippo Valsorda for the helpful discussions. This work has been supported by a Singapore Ministry of Education (MoE) Tier 3 grant "Automated Program Repair", MOE-MOET32021-0001. The author declares a non-financial interest as a developer of Gobra.

References

[1] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. 2017. Jasmin: High-Assurance and High-Speed Cryptography. In *CCS*. ACM, 1807–1823. doi:10.1145/3133956.3134078

[2] José Bacelar Almeida, Denis Firsov, Tiago Oliveira, and Dominique Unruh. 2023. Schnorr Protocol in Jasmin. Cryptology ePrint Archive, Paper 2023/752. https://eprint.iacr.org/2023/752

[3] Linard Arquint. 2026. crypto/internal/fips140/bigmod: Fix 'extendedGCD' Implementation Mismatch. https://go-review.googlesource.com/c/go/+/770380

[4] Linard Arquint. 2026. Verification of extendedGCD. https://github.com/arquintl/go-gcd

[5] Linard Arquint. 2026. Verification of extendedGCD – Continuous Verification. https://github.com/arquintl/go-gcd/actions/workflows/verify-gcd.yml

[6] Linard Arquint, Samarth Kishor, Jason R. Koenig, Joey Dodds, Daniel Kroening, and Peter Müller. 2026. The Secrets Must Not Flow: Scaling Security Verification to Large Codebases. In *S&P*. IEEE, 492–511. doi:10.1109/SP63933.2026.00026 To appear.

[7] Linard Arquint, Malte Schwerhoff, Vaibhav Mehta, and Peter Müller. 2023. A Generic Methodology for the Modular Verification of Security Protocol Implementations. In *CCS*. ACM, 1377–1391. doi:10.1145/3576915.3623105

[8] Linard Arquint, Felix A. Wolf, Joseph Lallemand, Ralf Sasse, Christoph Sprenger, Sven N. Wiesner, David A. Basin, and Peter Müller. 2023. Sound Verification of Security Protocols: From Design to Interoperable Implementations. In *S&P*. IEEE, 1077–1093. doi:10.1109/SP46215.2023.10179325

[9] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella-Béguelin. 2011. Computer-Aided Security Proofs for the Working Cryptographer. In *CRYPTO (LNCS, Vol. 6841)*. Springer, 71–90. doi:10.1007/978-3-642-22792-9_5

[10] David Benjamin. 2018. Add a Constant-Time Binary Extended GCD Algorithm. https://github.com/mit-plv/fiat-crypto/pull/333

[11] Barry Bond, Chris Hawblitzel, Manos Kapritsos, K. Rustan M. Leino, Jacob R. Lorch, Bryan Parno, Ashay Rane, Srinath T. V. Setty, and Laure Thompson. 2017. Vale: Verifying High-Performance Cryptographic Assembly Code. In *USENIX Security Symposium*. USENIX Association, 917–934.

[12] John Boyland. 2003. Checking Interference with Fractional Permissions. In *SAS (LNCS, Vol. 2694)*. Springer, 55–72. doi:10.1007/3-540-44898-5_4

[13] Cryspen. 2020. libcrux - A High-Assurance Cryptographic Library in Rust. https://github.com/cryspen/libcrux

[14] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. 2019. Simple High-Level Code for Cryptographic Arithmetic - With Proofs, Without Compromises. In *S&P*. IEEE, 1202–1219. doi:10.1109/SP.2019.00005

[15] Go developers. 2026. testing.B.Loop Documentation. https://pkg.go.dev/testing#B.Loop

[16] K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *LPAR (Dakar) (LNCS)*. Springer, 348–370. doi:10.1007/978-3-642-17511-4_20

[17] Alfred Menezes, Paul C. van Oorschot, and Scott A. Vanstone. 1996. *Handbook of Applied Cryptography*. CRC Press. doi:10.1201/9781439821916

[18] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2016. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *VMCAI (LNCS, Vol. 9583)*. Springer, 41–62. doi:10.1007/978-3-662-49122-5_2

[19] National Institute of Standards and Technology. 2019. *FIPS PUB 140-3: Security Requirements for Cryptographic Modules*. Federal Information Processing Standards Publication 140-3. National Institute of Standards and Technology. doi:10.6028/NIST.FIPS.140-3

[20] João C. Pereira, Tobias Klenze, Sofia Giampietro, Markus Limbeck, Dionysios Spiliopoulos, Felix Wolf, Marco Eilers, Christoph Sprenger, David A. Basin, Peter Müller, and Adrian Perrig. 2025. Protocols to Code: Formal Verification of a Secure Next-Generation Internet Router. In *CCS*. ACM, 1469–1483. doi:10.1145/3719027.3765104

[21] Jade Philipoom. 2023. Add a Constant-Time Binary Extended GCD Algorithm. https://github.com/mit-plv/fiat-crypto/pull/1597

[22] Jonathan Protzenko, Bryan Parno, Aymeric Fromherz, Chris Hawblitzel, Marina Polubelova, Karthikeyan Bhargavan, Benjamin Beurdouche, Joonwon Choi, Antoine Delignat-Lavaud, Cédric Fournet, Natalia Kulatova, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy, Christoph M. Wintersteiger, and Santiago Zanella Béguelin. 2020. EverCrypt: A Fast, Verified, Cross-Platform Cryptographic Provider. In *S&P*. IEEE, 983–1002. doi:10.1109/SP40000.2020.00114

[23] John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS*. IEEE, 55–74. doi:10.1109/LICS.2002.1029817

[24] Junyang Shao. 2025. More Predictable Benchmarking with testing.B.Loop. https://go.dev/blog/testing-b-loop

[25] Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean Karim Zinzindohoue, and Santiago Zanella Béguelin. 2016. Dependent Types and Multi-Monadic Effects in F*. In *POPL*. ACM, 256–270. doi:10.1145/2837614.2837655

[26] The Formosa team. [n. d.]. Formosa Crypto. https://formosa-crypto.org

[27] Filippo Valsorda. 2024. crypto/internal/fips140/bigmod: Add Nat.InverseVarTime. https://go-review.googlesource.com/c/go/+/632415

[28] Filippo Valsorda. 2025. crypto/internal/fips140/bigmod: Explicitly Clear Expanded Limbs on Reset. https://go-review.googlesource.com/c/go/+/655056

[29] Felix A. Wolf, Linard Arquint, Martin Clochard, Wytse Oortwijn, João Carlos Pereira, and Peter Müller. 2021. Gobra: Modular Specification and Verification of Go Programs. In *CAV (LNCS, Vol. 12759)*. Springer, 367–379. doi:10.1007/978-3-030-81685-8_17

[30] Jean Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. 2017. HACL*: A Verified Modern Cryptographic Library. In *CCS*. ACM, 1789–1806. doi:10.1145/3133956.3134043