Google Go Cryptographic Libraries

Security Assessment

March 27, 2025

Prepared for:
Roland Shoemaker
Google

Prepared by: **Scott Arciszewski, Opal Wright, and Joop van de Pol**

Table of Contents

Project Summary

Contact Information

The following project manager was associated with this project:

Jeff Braswell, Project Manager
jeff.braswell@trailofbits.com

The following engineering director was associated with this project:

Jim Miller, Engineering Director, Cryptography
james.miller@trailofbits.com

The following consultants were associated with this project:

Scott Arciszewski, Consultant
scott.arciszewski@trailofbits.com

Joop van de Pol, Consultant
joop.vandepol@trailofbits.com

Opal Wright, Consultant
opal.wright@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
January 6, 2025	Pre-project kickoff call
January 13, 2025	Status update meeting #1
January 21, 2025	Status update meeting #2
January 27, 2025	Status update meeting #3
February 4, 2025	Delivery of report draft
February 5, 2025	Report readout meeting
March 27, 2025	Delivery of final comprehensive report

Executive Summary

Engagement Overview

Google engaged Trail of Bits to review the security of the Go cryptography libraries. This includes the cryptography implementations in the FIPS-140 module.

A team of three consultants conducted the review from January 6 to February 3, 2025, for a total of 12 engineer-weeks of effort. Our testing efforts focused on identifying cryptographic weaknesses, such as side-channel attacks. With full access to source code and documentation, we performed static and dynamic testing of the codebase, using automated and manual processes.

Observations and Impact

We reviewed the Go cryptography libraries, including specific public modules in the `crypto` package, its corresponding internal features, and the new FIPS-140 module.

We generally found the cryptography algorithms in scope to be implemented securely. We identified six issues, ranging from a memory safety issue in the C bindings to BoringSSL (TOB-GOCL-3) to missed opportunities to provide stronger security against timing side channels (TOB-GOCL-1, TOB-GOCL-6, and TOB-GOCL-2, a.k.a., CVE-2025-22866), as well as specification nits regarding NIST standards (TOB-GOCL-4, TOB-GOCL-5).

Recommendations

Based on the findings identified during the security review, Trail of Bits recommends that Google take the following steps:

- **Remediate the findings disclosed in this report.** These findings should be addressed as part of a direct remediation or any refactor that may occur when addressing other recommendations.
- **Clearly document user requirements for FIPS compliance.** For instance, include the SHA-256 input length limitation in the module documentation.
- **Consider adding additional FIPS functionality.** The SHA-3 module includes support for the cSHAKE functions outlined in NIST SP 800-185. Including FIPS-certified KMAC, TupleHash, and ParallelHash implementations could help FIPS users avoid several common cryptographic mistakes (see our blog post "YOLO is not a valid hash construction" for examples).

Finding Severities and Categories

The following tables provide the number of findings by severity and category.

EXPOSURE ANALYSIS

Severity	*Count*
High	0
Medium	0
Low	1
Informational	5
Undetermined	0

CATEGORY BREAKDOWN

Category	*Count*
Cryptography	5
Timing	1

Project Goals

The engagement was scoped to provide a security assessment of the Go cryptography libraries. Specifically, we sought to answer the following non-exhaustive list of questions:

- Can any inputs result in incorrect behavior for any of the cryptographic algorithms in scope?
- Are any dangerous features exposed publicly to users of the Go cryptography libraries? If so, are they obvious or astonishing?
- Does the runtime of any algorithm depend on the values of a cryptographic secret?
- Does the various platform-specific assembly code correctly implement the same algorithm as the generic implementation, and without introducing any side-channel leakage?
- Are constant-time hardware-accelerated implementations of certain algorithms (i.e., AES) always used when they are available?
- Does the interface between Go and C introduce any memory safety issues, and if so, can they be exploited in practice?
- Are only `math/big` functions that warn about cryptographic usage reachable from non-deprecated cryptography functions?

Project Targets

The engagement involved a review and testing of the following target. Please note that only specific modules were in scope of the review.

Go crypto/...

Repository	https://github.com/golang/go
Version	a76cc5a4ecb004616404cac5bb756da293818469
Type	Cryptographic library
Platform	amd64, arm64, ppc64, and s390x

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches included the following:

- Manual code review of the following modules:
  - `crypto/aes`
  - `crypto/cipher`
  - `crypto/ecdh`
  - `crypto/ecdsa`
  - `crypto/elliptic`
  - `crypto/hkdf`
  - `crypto/hmac`
  - `crypto/internal/bigmod`
  - `crypto/internal/hpke`
  - `crypto/internal/mlkem768`
  - `crypto/internal/nistec`
  - `crypto/md5`
  - `crypto/pbkdf2`
  - `crypto/rand`
  - `crypto/rsa`
  - `crypto/sha1`
  - `crypto/sha3`
  - `crypto/sha256`
  - `crypto/sha512`
- Static code analysis using Semgrep and CodeQL

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. The following list outlines the coverage limitations of the engagement and indicates system elements that may warrant further review:

- The deprecated elliptic package was not prioritized for deeper analysis related to implementations for custom (i.e., non-NIST) curves and may not provide constant-time implementations.
- Automated constant-time verification (such as with dudect) was not in scope for this assessment.
- Other modules (x509, tls) that make heavy use of the crypto module were not in scope.

Summary of Findings

The table below summarizes the findings of the review, including details on type and severity.

ID	Title	Type	Severity
1	Fiat conversion from bytes to field elements is not constant time	Cryptography	**Informational**
2	P-256 conditional negation is not constant time in PowerPC assembly	Cryptography	**Informational**
3	Custom finalizer may free memory at the start of a C function call using this memory	Timing	**Low**
4	The CTR-DRBG module presents multiple misuse risks	Cryptography	**Informational**
5	PBKDF2 does not enforce output length limitations	Cryptography	**Informational**
6	Timing leak in edwards25519 Scalar.SetCanonicalBytes	Cryptography	**Informational**

Detailed Findings

1. Fiat conversion from bytes to field elements is not constant time

Severity: **Informational**	Difficulty: **High**
Type: Cryptography	Finding ID: TOB-GOCL-1
Target: `src/crypto/internal/fips140/nistec/fiat/{generate.go, p256.go, p384.go, p521.go}`	

Description

The generic (i.e., non-assembly) implementation of elliptic curve cryptography for NIST curves relies on code generated from the `fiat-crypto` package to do curve operations. The implementation additionally uses generated Go code to convert byte buffers into field elements, which is used when constructing the coordinates elliptic curve point objects for most elliptic curve operations (such as ECDH, point addition and doubling, and scalar multiplication).

This conversion includes a byte-wise comparison between the coordinate(s) and the field prime minus one starting from the most significant byte. This comparison is stopped as soon as one byte of the coordinate is less than the corresponding byte of the field prime minus one, as shown in the following figure.

```
for i := range v {
	if v[i] < minusOneEncoding[i] {
		break
	}
	if v[i] > minusOneEncoding[i] {
		return nil, errors.New("invalid P256Element encoding")
	}
}
```

Figure 1.1: The comparison between coordinate bytes and field prime stops early (go/src/crypto/internal/fips140/nistec/fiat/p256.go#L81–L88)

This is not constant time in the input coordinates. In the vast majority of cases, elliptic curve input points are non-sensitive. However, there are some cases, such as the PACE protocol and some oblivious transfer protocols (such as Endemic OT), where the input point is sensitive and where leaking information on the input point may break the security of the scheme.

Exploit Scenario

A Go crypto library user uses the ECDH functionality to implement Endemic OT on a platform that relies on the fiat implementation. An attacker who participates as the sender crafts two public keys, where one has an x-coordinate whose most significant bytes agree with the field prime and the other has an x-coordinate whose first byte is less than the first byte of the field prime. The execution time of the party that acts as receiver leaks information on their choice bit.

Recommendations

Short term, process all input bytes when converting them to a field element. Alternatively, document that the relevant elliptic curve APIs do not consider input points to be sensitive.

2. P-256 conditional negation is not constant time in PowerPC assembly

Severity: **Informational**	Difficulty: **High**
Type: Cryptography	Finding ID: TOB-GOCL-2
Target: `src/crypto/internal/fips140/nistec/p256_asm_ppc64le.s`	

Description

The `p256NegCond` function takes a field element and a condition, and negates the field if the condition is non-zero. In the assembly implementation for the 64-bit PowerPC architecture, this function returns early if the condition is zero, as shown in the following figure.

```
// func p256NegCond(val *p256Point, cond int)
TEXT ·p256NegCond(SB), NOSPLIT, $0-16
	MOVD val+0(FP), P1ptr
	MOVD $16, R16

	MOVD cond+8(FP), R6
	CMP  $0, R6
	BC   12, 2, LR      // just return if cond == 0

	MOVD $p256mul<>+0x00(SB), CPOOL
```

Figure 2.1: Conditional negation returns early if the condition is zero (go/src/crypto/internal/fips140/nistec/p256_asm_ppc64le.s#L129–L138)

This function is used in scalar multiplication where the condition corresponds to one of the scalar bits. In the `p256BaseMult` function, this causes a timing difference based on a single scalar bit (the sixth least significant bit), whereas in the `p256ScalarMult` function this causes timing differences based on every fifth scalar bit.

Exploit Scenario

An attacker measures the global execution time of a scalar multiplication operation (not with the base point). They use this to get information on the combined Hamming weight of every fifth bit. Note that this is only a partial attack, since the obtained information is not sufficient to recover the key.

Recommendations

Short term, make the implementation of the `p256NegCond` function constant time by using the VSEL instruction, similar to the ppc64 `p256MovCond` implementation and the s390x `p256NegCond` implementation.

3. Custom finalizer may free memory at the start of a C function call using this memory

Severity: **Low**	Difficulty: **High**
Type: Timing	Finding ID: TOB-GOCL-3
Target: `src/crypto/internal/boring/ecdh.go`	

Description

The library offers two backing implementations for ECDH using the NIST curves P-256, P-384, and P-521: the `fips140` package, which is implemented in Go (with some optional assembly for P-256); and the `boring` package, which relies on `cgo` to call into the BoringSSL library written in C.

In the `boring` package, the private and public keys are maintained using Go structs that contain C pointers for the `key` field (as well as the `group` field of the public key). When creating new keys, they are provided with a custom finalizer that calls the corresponding function from the BoringSSL library to free the memory of the C pointer in the `key` field.

Later, the keys are used by passing the corresponding C pointer into the BoringSSL C function. If this comprises the last reference to the key Go object in the Go code, the garbage collector is free to collect this object, calling the finalizer. If this happens due to a coincidence in timing, it may trigger a use-after-free in the C function.

To prevent this, it is necessary to call `runtime.KeepAlive` on the Go object at any moment after the call to the C function, as shown in the following figure.

```
func (k *PrivateKeyECDH) PublicKey() (*PublicKeyECDH, error) {
	defer runtime.KeepAlive(k)

	group := C._goboringcrypto_EC_KEY_get0_group(k.key)
```

Figure 3.1: The public key object is kept alive (go/src/crypto/internal/boring/ecdh.go#L101–L104)

However, the ECDH function does not include such a function call, as shown in the following figure.

```
func ECDH(priv *PrivateKeyECDH, pub *PublicKeyECDH) ([]byte, error) {
	group := C._goboringcrypto_EC_KEY_get0_group(priv.key)
	if group == nil {
		return nil, fail("EC_KEY_get0_group")
```

```
        }
        privBig := C._goboringcrypto_EC_KEY_get0_private_key(priv.key)
        if privBig == nil {
                return nil, fail("EC_KEY_get0_private_key")
        }
        pt := C._goboringcrypto_EC_POINT_new(group)
        if pt == nil {
                return nil, fail("EC_POINT_new")
        }
        defer C._goboringcrypto_EC_POINT_free(pt)
        if C._goboringcrypto_EC_POINT_mul(group, pt, nil, pub.key, privBig, nil) == 0
{
                return nil, fail("EC_POINT_mul")
        }
        out, err := xCoordBytesECDH(priv.curve, group, pt)
        if err != nil {
                return nil, err
        }
        return out, nil
}
```

Figure 3.2: The ECDH function does not explicitly keep the public and private key alive (go/src/crypto/internal/boring/ecdh.go#L140–L162)

The call to `xCoordBytesECDH` ensures that private key `priv` is not collected, but the public key `pub` may be collected at the beginning of the call to `C._goboringcrypto_EC_POINT_mul`.

Exploit Scenario

This issue may arise during normal operations, causing a crash. Alternatively, an attacker may try to influence the garbage collection and exploit the use-after-free.

Recommendations

Short term, add (deferred) `KeepAlive` calls for the ECDH function.

Long term, consider adding a wrapper that forces such calls, such as the `withKey` function from the RSA code in the `boring` package.

4. The CTR-DRBG module presents multiple misuse risks

Severity: **Informational**	Difficulty: **N/A**
Type: Cryptography	Finding ID: TOB-GOCL-4
Target: `src/crypto/internal/fips140/drbg/ctrdrbg.go`	

Description

The CTR-DRBG implementation uses the initialization and reseeding algorithms specified in NIST SP 800-90A, sections 10.2.1.3.1 and 10.2.1.4.1, respectively. These sections do not use the derivation function for preprocessing seed material.

If the derivation function is not used and the DRBG state is compromised, an attack identified by Woodage and Shumow (included in the References section below) allows recovery of subsequent entropy inputs. Applied to the implementation here, the attack would require knowing 256 of the 384 input bits. This sounds improbable, but the possibility of a fixed 256-bit entropy prefix is increased by the lack of personalization strings and the fixed length of additional inputs.

NIST SP 800-90A specifies that the additional inputs can include sensitive values like key material, shared secrets, or password information. This means users of the `drbg` module could expose potentially sensitive values in the output of the DRBG.

The FIPS-compliant PRNG interface exposed in the `rand` module is not subject to this attack, as the first 128 bits of the 256-bit prefix are not fixed, and the final 128 bits are fixed to zero and do not reveal any sensitive information.

Additionally, the CTR-DRBG implementation does not support personalization strings. For the PRNG interface in the `rand` module, this does not present a security problem. However, failure to personalization strings may cause the implementation to be flagged as non-compliant or only partially compliant.

Recommendations

Short term, implement personalization string support, and clearly document that the current implementation relies on application-specific circumstances to maintain security. If personalization string support is not desired, document it as a deviation from SP 800-90A.

Long term, consider updating the CTR-DRBG implementation to use the derivation function.

References

- An Analysis of the NIST SP 800-90A Standard by Joanne Woodage and Dan Shumow

5. PBKDF2 does not enforce output length limitations

Severity: **Informational**	Difficulty: **N/A**
Type: Cryptography	Finding ID: TOB-GOCL-5
Target: `src/crypto/internal/fips140/pbkdf2/pbkdf2.go`	

Description

The PBKDF2 implementation does not enforce any output length checks. Section 5.3 of NIST SP 800-132 states the output of PBKDF2 cannot be longer than $2^{32} - 1$ hash blocks.

A comment in the PBKDF2 implementation states that it is based on RFC 8018, rather than NIST SP 800-132. RFC 8018 states:

> The length of the derived key is essentially unbounded.

However, the algorithm specification still includes the output length check.

Recommendations

Shore term, implement the output length check.

Long term, attempt to map all steps of the algorithm specification to the code.

6. Timing leak in edwards25519 Scalar.SetCanonicalBytes

Severity: **Informational**	Difficulty: **N/A**
Type: Cryptography	Finding ID: TOB-GOCL-6
Target: `src/crypto/internal/fips140/edwards25519/scalar.go`	

Description

The public `SetCanonicalBytes` method in the `edwards25519` Scalar struct calls a variable-time private `isReduced` method, which compares the scalar bytes against one less than the group order and aborts early.

```
func (s *Scalar) SetCanonicalBytes(x []byte) (*Scalar, error) {
	if len(x) != 32 {
		return nil, errors.New("invalid scalar length")
	}
	if !isReduced(x) {
		return nil, errors.New("invalid scalar encoding")
	}
```

Figure 6.1: The `SetCanonicalBytes` function (go/src/crypto/internal/fips140/edwards25519/scalar.go#L155–L170)

This leaks information about the most significant bytes of the scalar through its runtime.

```
func isReduced(s []byte) bool {
	if len(s) != 32 {
		return false
	}
	for i := len(s) - 1; i >= 0; i-- {
		switch {
		case s[i] > scalarMinusOneBytes[i]:
			return false
		case s[i] < scalarMinusOneBytes[i]:
			return true
		}
	}
	return true
}
```

Figure 6.2: The `isReduced` function (go/src/crypto/internal/fips140/edwards25519/scalar.go#L177–L191)

This leakage is not adaptive, and this code is used elsewhere in the Go crypto library with only public inputs (i.e., to ensure that a public value is smaller than the group order, which prevents signature malleability).

Because this is a public method, another application could use it and unexpectedly leak some information about its secret key. However, since this comparison starts with the most significant byte due to the order of the `edwards25519` group, this comparison will proceed past the first byte with probability of only 2^{-127}, so this is a negligible probability.

Recommendations

Short term, either implement the private function in constant-time (example below) or update the public method's name to indicate that it is deliberately not constant-time (e.g., `SetCanonicalBytesVarTime`).

```
func isReduced(s []byte) bool {
	if len(s) != 32 {
		return false
	}

	// Adopted from libsodium's compare function
	gt := uint16(0)
	eq := uint16(1)
	for i := len(s) - 1; i >= 0; i-- {
		b1 := uint16(s[i])
		b2 := uint16(scalarMinusOneBytes[i])
		gt = gt | (((b2 - b1) >> 8) & eq)
		eq = eq & ((b2 ^ b1) - 1) >> 8
	}
	//   {gt, eq} := {1, 0} means left > right, which will return 1
	//   {gt, eq} := {0, 1} means they were equal, which will return 0
	//   {gt, eq} := {0, 0} means left < right, which will return -1
	// return (gt + gt + eq - 1)
	// {1, 1} is not possible to reach through this logic.
	// We are only interested in cases where left <= right, so we only need gt
after the loop.
	// Therefore, we just need to return gt == 0.
	return gt == 0
}
```

Figure 6.3: An updated `isReduced` function without branches or early aborts

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories	
Category	**Description**
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

B. Scalar Multiplication Analysis

This section describes the scalar multiplication analysis we conducted during the audit.

Introduction

The Go crypto library contains a dedicated implementation for elliptic curve scalar multiplication on the NIST P-256 curve for several platforms with support for vector instructions. The scalar multiplication implementation includes point doubling and addition based on incomplete formulae. Here, incomplete means that there are some input points (problematic points) that are not properly handled by this formula. This appendix analyzes whether various implementations create such problematic points.

Scalar Multiplication

The implementations use a fixed-window approach, where the scalar bits in each window are encoded using Booth encoding, i.e., a w-bit number is represented as a signed $(w - 1)$ -bit number, where w is the size of the window. Specifically, a number $x \in [0, 2^w)$ is represented as $y \in [-2^{w-1}, 2^w)$, where $x = y \bmod 2^w$. In other words, if y is positive, then $x = y$, whereas if y is negative, then $x = 2^w + y$. When processing multiple windows during scalar multiplication, this term 2^w (the borrow) needs to be accounted for in the next window.

This windowed approach requires that certain multiples of the point to be multiplied are precomputed. Because point negation is easy, it is only necessary to store positive multiples, which saves storage. For scalar multiplication with the fixed base point, the window size is six, and all shifted windows of the base point are precomputed and stored in a table before compilation. For scalar multiplication with a general input point, the window size is five, and only a single window is precomputed and stored in a table.

Regardless, scalar multiplication now consists of a loop where in each iteration:

- there is a point $x * G$ in the accumulator, and
- the implementation adds a point $y * G$ (which is selected from a table).

Here, x is the Booth encoding of the part of the scalar bits that have been processed so far (i.e., if i windows have been processed so far, x represents a number of $i \cdot w$ bits). On the other hand, y is the Booth encoding of the scalar bits in the next window to be processed (i.e., y represents a number of w bits).

After this addition, the accumulator will contain the point $(x + y) * G$, which will represent $(i + 1) \cdot w$ bits. Note that this point may need to be doubled a number of times before the next iteration, depending on the implementation. After the last iteration, the accumulator will contain the point $d * G$ where d is the input scalar.

Problems arise in the addition step when $x * G = \pm y * G$, $x = 0$, and/or $y = 0$ due to the usage of incomplete formulae. It is important to note that the problematic points of $x = 0$ and/or $y = 0$ are correctly handled by the implementation. This means the analysis can focus on $x * G = \pm y * G$ for non-zero x, y, which is equivalent to stating $x \pm y = 0 \bmod N$. Furthermore, the callers of the scalar multiplication functions guarantee that the input scalar is in the interval $[0, N)$.

Now, the analysis of problematic points can be sketched as follows:

- The numbers x and y always represent disjoint windows. As a result, we have that $x \pm y = 0$ implies that $x = y = 0$. As mentioned before, the implementations properly handle this case.
- The numbers x and y together represent $(i + 1) \cdot w$ bits, and never more than $\lceil \log_2(N) \rceil$ bits. This means that, no matter what, $|x \pm y| < 2 \cdot N$, and in many cases it is easy to see that $|x \pm y| < N$. This especially holds in all iterations except the last one, because x and y do not yet represent sufficiently many bits to represent numbers that can wrap around modulo N. Note that $|x \pm y| < N$ and $x \pm y = 0 \bmod N$ imply that $x = y = 0$ due to the previous bullet.
- Therefore, the analysis can focus on the last iteration to see whether problematic points can arise. In the last iteration, the case where $x + y = N$ is ruled out by the fact that $x + y = d \in [0, N)$.

As mentioned, there are two implementations: one for scalar multiplication with the fixed generator point, and one for scalar multiplication with a provided input point. As the implementation for the fixed generator has all shifted windows precomputed, it does not matter whether the algorithm works right-to-left (i.e., starting from the LSB side of the scalar and working up to the MSB side) or left-to-right, since the Booth encoding efficiently supports both options. The current implementation uses right-to-left, but a reverted commit changed this to left-to-right (this was reverted due to an issue with problematic points). The scalar multiplication with a provided input point precomputes only a single window, which works only with a left-to-right implementation for efficiency reasons (otherwise every iteration would include doubling the precomputed points several times).

Scalar Multiplication with Generator, Right-to-Left Implementation

In the `ScalarBaseMult` right-to-left implementation, we have that:

- The window size is 6, but some results below generalize to other window sizes;
- $x \in [-2^{6i-1}, 2^{6i-1} - 1]$ for iteration i; and
- $y = z \cdot 2^{6i}$ for $z \in [-32, 32]$ for iteration i (except for the last iteration ($i = 42$), where $z \in [0, 16]$).

Note that $x \pm y = 0 \Leftrightarrow x = y = 0$, because $|x| \leq 2^{6i-1} < 2^{6i}$, and $y \neq 0 \Leftrightarrow |y| \geq 2^{6i}$ (disjoint windows).

There is a problem with the incomplete formulae when $x * G = \pm y * G$, or equivalently, if $x \pm y = 0 \bmod N$. Rewriting this gives that $x \pm y = kN$ for some k. Because

$$|x| < 2^{251},\ |y| \leq 2^{256},\ |x \pm y| < 2N,$$

which means $k \in \{-1, 0, 1\}$.

$k = 0$ implies that $x = y = 0$, as described above. Those cases are handled by the implementation: both the variables `zero` and `sel` will be 0, and a garbage point is produced. This garbage point is later discarded and replaced by the correct point when possible.

$k = \pm 1$ implies that $x \pm y = \pm N$ (where all four combinations are potentially valid). Now, $|x + y| < |x| + |y| \leq 2^{6i-1} + 2^{6i+5}$. For the iterations $i < 42$, this is equal to $2^{245} + 2^{251} < 2^{252} < N$.

So, the only iteration where the previous equality can hold is $i = 42$. In this iteration, z (and therefore y) cannot be negative, which rules out the cases $x + y = -N$ and $x - y = N$.

For $x + y = N$, the binary representation of the scalar is equal to the binary representation of N, which cannot happen because there is a modular reduction $\bmod N$ before the scalar multiplication.

For $x - y = -N$, we can rewrite this as $y - x = N$. We have $x \in [-2^{251}, 2^{251} - 1]$, and $y = z \cdot 2^{252}$ for $z \in [0, 16]$.

- If $z = 16$, then y got a borrow, which implies that x was negative, which implies that $y - x = y + |x| > 2^{256} > N$, which is a contradiction.
- If $z < 16$, then $y \leq (16 - 1) \cdot 2^{252} = 2^{256} - 2^{252}$ and $|y - x| < y + |x| \leq 2^{256} - 2^{252} + 2^{251} = 2^{256} - 2^{251} < N$, which is a contradiction.

We conclude that there is no scalar that results in the processing of problematic points by `ScalarBaseMult` as currently implemented.

Scalar Multiplication with Generator, Left-to-Right Implementation

In the `ScalarBaseMult` left-to-right implementation, we first consider some preliminaries.

- We write N in base $2^6 = 64$, with digits n_j (i.e., $N = \sum_{j=0}^{42} n_j \cdot 2^{6j}$, where $n_j \in [0, 63]$ and $n_{42} \in [0, 15]$ are known constants).
- We define $N_i^H := \sum_{j=i}^{42} n_j \cdot 2^{6j}$ and $N_i^L := \sum_{j=0}^{i-1} n_j \cdot 2^{6j}$ such that $N = N_i^H + N_i^L$

Now, we have that:

- The window size is 6, but some results below generalize to other window sizes;
- $x = t \cdot 2^{6i}$ such that $0 \leq x \leq N_i^H + 2^{6i}$ for iteration i (we consider that the iterations to go down from 42 to 1); and
- $y = z \cdot 2^{6i-6} * G$, where $z \in [-32, 32]$ for iteration i (except for the last iteration ($i = 1$), where $z \in [-32, 31]$).

Note that x is non-negative and bounded by the high bits of N, with the exception that it can pull in a borrow from the next window, in which case $x = N_i^H + 2^{6i}$. In any case, we have the same as before (disjoint windows): $x \pm y = 0 \Leftrightarrow x = y = 0$, because $|y| \leq 2^{6i-1} < 2^{6i}$, and $x \neq 0 \Leftrightarrow |x| \geq 2^{6i}$.

Again, there is a problem with the incomplete formulae when $x * G = \pm y * G$, or equivalently, if $x \pm y = 0 \bmod N$. Rewriting this gives that $x \pm y = kN$ for some k. Since x is always positive, and since $x \leq N_i^H + 2^{6i} < N + 2^{6i}$ and $|y| = |z \cdot 2^{6i-6}| \leq 2^{6i-1} < 2^{6i}$, we have that $k \in \{0, 1\}$.

$k = 0$ implies that $x = y = 0$, as described above. Those cases are handled by the implementation as previously discussed.

Note that $x \bmod 2^{6i-6} = y \bmod 2^{6i-6} = 0$, but $N \bmod 2^{6i-6} = N_{i-1}^L$, which is nonzero unless $i = 1$, so $x \pm y = N$ implies $i = 1$.

Since the sixth LSB of N is zero, we cannot have $x = N_1^H + 2^6$, because x cannot pull in a borrow in the previous iteration. Therefore, we have that $x = t \cdot 2^6 \leq N_1^H$.

- Now $x < N_1^H$ implies that $N - x \geq 2^6$, and since $|y| \leq 2^5$, we have that $|y| < N - x$, which is a contradiction.
- If $x = N_1^H = N - 17$, this implies that y = -17. This corresponds to the problematic scalar $N - 34$ that was reported for the reverted commit. Note that this only works because the booth encoding of $N - 34$ results in $x = N_1^H = N - 17$.

According to the above analysis, this is the only scalar that can cause problematic points.

Scalar Multiplication with Input Point, Left-to-Right Implementation

Now, repeating this for the `ScalarMult` left-to-right implementation:

- G is now P.

- The window size is 5, and the shifted windows are not precomputed. This means that the accumulator has to be doubled every iteration. Therefore:
  - We write N in base $2^5 = 32$, with digits n_j (i.e., $N = \sum_{j=0}^{51} n_j \cdot 2^{5j}$, where $n_j \in [0, 31]$ and $n_{51} \in [0, 1]$ are known constants.
  - We redefine $N_i^H := \sum_{j=i}^{51} n_j \cdot 2^{5(j-i)}$ and $N_i^L := \sum_{j=0}^{i-1} n_j \cdot 2^{5j}$ such that $N = N_i^H \cdot 2^{5i} + N_i^L$.
- $x = t \cdot 2^5$ such that $t \in [0, N_i^H + 1]$ for iteration i (we consider that the iterations go from 51 down to 1).
- $y \in$ [-16, 16] (except for the last iteration (i=1), where $y \in$ [-16, 15]).

More generally, there is a problem with the incomplete formulae when $x * P = \pm y * P$, or equivalently, if $x \pm y = 0 \bmod N$. Rewriting this gives that $x \pm y = kN$ for some k. Since x is always nonnegative, and since $x \leq 2^5 \cdot (N_i^H + 1) < 2^{256-5i+5}$ and $|y| \leq 2^4$, we have that $i > 1$ implies $k = 0$ (which implies that $x = y = 0$ like before). If $i = 1$, then $x \leq 2^5 \cdot (N_1^H + 1) = N + 15$ and $y \in$ [-16, 15], so $x \pm y = kN$ implies $k \in \{0, 1\}$. Again, $k = 0$ implies that $x = y = 0$ like before, which leaves $k = 1$.

There are three cases:

- $x = 2^5 \cdot (N_1^H + 1) = N + 15$: Unlike the 6-bit window case, this is possible, because the fifth LSB of N is one. So a corresponding $y = \pm 15$ would lead to a problematic point. The value $y = 15$ is not possible, because then x would not have obtained the borrow in the previous iteration. The value $y =$ -15 results in a problematic point, but this combination of x and y corresponds to the input scalar of N which would get reduced to zero before the scalar multiplication function call.
- $x = 2^5 \cdot N_1^H = N - 17$: Note that $2^5 \cdot N_1^H = N - 17$, even though the window size has changed. The reason is that the sixth LSB of N is zero. However, unlike the 6-bit window case, the number $N - 34$ does not result in an accumulated point of $x = N - 17$ at iteration 1, so this scalar does not result in problematic points.
- $x < 2^5 \cdot N_1^H$: This implies $x \leq 2^5 \cdot N_1^H - 2^5 < N - 2^5$, and because $y \leq 2^4$, $|x \pm y| \leq x + |y| < N$ which is a contradiction.

So, we conclude that there are no scalars that lead to the processing of problematic points by the `ScalarMult` function.

C. Static Code Analysis

This appendix describes the setup of the static analysis tools we used during this audit.

C.1 Semgrep

After installing Semgrep and creating a directory of rules, we ran the following command to the codebase:

```
semgrep --metrics=off --config=r/all
--config=${SEMGREP_RULES_DIR} --pro  --exclude=*.sarif --sarif .
> semgrep.sarif
```

We used a mix of public and private rules, which did not identify any significant findings.

C.2 CodeQL

Building the database

After compiling the compiler using `make.bash`, it is possible to generate a CodeQL database using the command:

```
codeql database create dbname.db --language=go
--command='./bin/go build -a std cmd'
```

In order to run queries on functionality in the boring package, it is necessary to build using the environmental flag `GOEXPERIMENT=boringcrypto`. This further requires environmental flags `CGO_ENABLED=1`, `GOOS=linux` (which requires environment variable `CC` to be set for cross-compilation if applicable).

Custom queries

Detecting functions from `math/big` used by `crypto`

Some parts of the Go standard library `crypto` package rely on the big integer package `math/big`. Important functions in this package contain a comment stating that they are used in cryptographic operations, and that changes need to be reviewed by a security expert.

```
func (x *Int) Sign() int {
	// This function is used in cryptographic operations. It must not leak
	// anything but the Int's sign and bit size through side-channels. Any
	// changes must be reviewed by a security expert.
```

Figure C.1: The Int.Sign function in the math/big package (<path>)

In order to determine whether this covers all functions in the `math/big` package, we created a custom CodeQL rule, as shown in figure C.2. It checks all paths starting from

non-deprecated externally available functions of the `crypto` package leading to functions in the `math/big` package that do not contain the comment.

The CodeQL rule detected the following `math/big.Int` functions:

- `Cmp`
- `FillBytes`
- `Int64`
- `IsInt64`
- `SetBits`
- `SetBytes`
- `SetUint64`
- `Sub`
- `Neg/Add` (via `cryptobyte.ReadASN1Integer`)

The following functions in the `crypto` package call one or more of these functions:

- `rand.Int`
- `rand.Prime`
- `ecdsa.PrivateKey.ECDH`
- `ecdsa.GenerateKey`
- `rsa.GenerateKey`
- `rsa.Decrypt`
  - `rsa.DecryptOAEP`
  - `rsa.DecryptPKCS1v15`
  - `rsa.DecryptPKCS1v15SessionKey`
- `rsa.Sign`
  - `rsa.SignPKCS1v15`
  - `rsa.SignPSS`
- `rsa.Validate`

- `rsa.Precompute`

Additionally, the following functions call `cryptobyte.ReadASN1Integer`, which calls `math/big.Int.Sub` and `math/big.Int.Neg`:

- `ecdsa.Sign`
- `ecdsa.Verify`
- `ecdsa.VerifyASN1`

Additional analysis of the functions involved did not reveal any security issues in these functions and the way they are used. For some callers, like RSA key generation, constant time operation is not relevant. For other callers, the functions are applied to public data.

The following figure shows a custom CodeQL rule to detect `math/big` functions that do not contain the warning called from non-deprecated `crypto` package functions:

```
/**
 * @name Non-deprecated cryptography functions should only call into marked math/big
functions
 * @kind path-problem
 * @problem.severity warning
 * @id go/custom/std-crypto-calls-big
 */

import go

/** Functions in the big package */
class BigFunction extends Function {
  BigFunction() { this.getPackage().getPath() = "math/big" }
}

/** Functions in the big package with the warning comment */
class WarnBigFunction extends BigFunction {
  Comment c;

  WarnBigFunction() {
    // The first line of the warning comment group matches this text
    c.getText() = " This function is used in cryptographic operations. It must not
leak" and
    // The comment group is located at the first line of the corresponding function
declaration
    c.getGroup().hasLocationInfo(this.getFuncDecl().getFile().getAbsolutePath(),
this.getFuncDecl().getLocation().getStartLine()+1,
c.getGroup().getLocation().getStartColumn(),
c.getGroup().getLocation().getEndLine(), c.getGroup().getLocation().getEndColumn())
  }
}

/** Functions in the big package without the warning comment */
class NonWarnBigFunction extends BigFunction {
  NonWarnBigFunction() {
    not this instanceof WarnBigFunction
  }
}

/** Functions in the crypto package */
class CryptoFunction extends Function {
// CryptoFunction () {this.getPackage().getName() in ["crypto", "aes", "boring",
"cipher", "des", "dsa", "ecdsa", "ed25519", "elliptic", "fips140", "hkdf", "hmac",
"md5", "mlkem", "pbkdf2", "rand", "rc4", "rsa", "sha1", "sha3", "sha256", "sha512",
"subtle", "tls", "x509"] }
  CryptoFunction () {
```

```
    // Only include packages starting with crypto
    this.getPackage().getPath().matches("crypto%")
  }
}

/** Deprecated functions, i.e. functions with the word deprecated in their doc
comment or 'Legacy' in their name */
class DeprecatedFunction extends Function {
  DeprecatedFunction() {
    // Deprecated functions have the word "Deprecated" in their doc comment

this.getFuncDecl().getDocumentation().getAComment().getText().matches("%Deprecated%"
)
    // We also exclude functions with Legacy in the name otherwise you get things
like legacy ECDSA signing which are not explicitly marked deprecated
    or this.getName().matches("%Legacy%")
  }
}

/** Non-deprecated functions */
class NonDeprecatedFunction extends Function {
  NonDeprecatedFunction () {
    not this instanceof DeprecatedFunction
  }
}

/** External functions of the crypto package that are not deprecated */
class ExternalNonDeprecatedCryptoFunction extends CryptoFunction,
NonDeprecatedFunction {
  ExternalNonDeprecatedCryptoFunction () {
    // Exclude internal packages
    not this.getPackage().getPath().matches("%internal%")
    // Exclude deprecated des/dsa/rc4 packages
    and not this.getPackage().getName() in ["des", "dsa", "rc4"]
    // Exclude deprecated elliptic package functions except PXXX curves
    and not (this.getPackage().getName() = "elliptic" and not this.getName() in
["P224", "P256", "P384", "P521"])
    // Only include exported functions that start with a capital letter
    and this.getName().charAt(0).isUppercase()
  }
}

/** Predicate that shows when a function calls a non-deprecated function */
query predicate edges(Function caller, NonDeprecatedFunction callee) {
  // The call to the callee is inside the function declaration of the caller
  callee.getACall().asExpr().getEnclosingFunction() = caller.getFuncDecl()
```

```
  // Now we need to eliminate strange interface matching by CodeQL
  and (
    // Either the caller imports the package of the callee
    exists (ImportSpec is | is.getFile() = caller.getFuncDecl().getFile() |
callee.getPackage().getPath() = is.getPath())
    or
    // Or the caller and callee are in the same package
    caller.getPackage() = callee.getPackage()
  )
  // We also eliminate when the caller is already a target function in the
  // math/big package because we are only interested in the first occurrence.
  // Comment out this line if you want to obtain all reachable functions.
  and not caller instanceof NonWarnBigFunction
}

// Query: start from an external non-deprecated crypto function and find
// a path of edges to a math/big function without the warning.
// We exclude starting functions in the x509 and tls packages, but those
// lines can be commented if you want to include them.
from ExternalNonDeprecatedCryptoFunction start, NonWarnBigFunction end
where edges+(start,end)
  and not start.getPackage().getPath().matches("%x509%")
  and not start.getPackage().getPath().matches("%tls%")
select end, start, end, "A non-deprecated function from the crypto package calls a
math/big function without the warning comment."
```

Figure C.2: Custom CodeQL rule to detect `math/big` functions that do not contain the warning called from non-deprecated `crypto` package functions

Detecting finding TOB-GOCL-3

The finding TOB-GOCL-3 relates to code with C bindings generated by `cgo` in the `boring` package. Specifically, there are several types with a custom finalizer that frees one of the fields of this type. When calling a C function with this field as an argument, the garbage collection might trigger at the start of this function call. The following CodeQL rule detects this. The query did not detect any instances besides the one reported in TOB-GOCL-3.

```
/**
 * @name Boring KeepAlive
 * @kind problem
 * @problem.severity warning
 * @id go/custom/boring-keepalive
 */

import go

/** A runtime.SetFinalizer() call to set a custom finalizer */
class RuntimeSetFinalizerCall extends CallExpr {
    RuntimeSetFinalizerCall() {
        // The target function of this call is from the runtime package
        this.getTarget().getPackage().getName() = "runtime"
        and
        // The target function of this call is called SetFinalizer
        this.getTarget().getName() = "SetFinalizer"
    }
}

/**
 * The Type that we are interested in: in the boring package with a custom finalizer
 * that calls some _free or _cleanup function
 */
class BoringTypeWithCF extends Type {
    RuntimeSetFinalizerCall setfinalcall;
    Function final;
    Field finalfield;
    CallExpr freecall;
    Variable v;
    DataFlow::ReadNode init_read;
    DataFlow::ReadNode free_read;
    DataFlow::Node field_base;


    BoringTypeWithCF() {
        // The package must be boring
        this.getEntity().getDeclaration().getFile().getPackageName() = "boring"
        and
        // The first argument of the SetFinalizer call is a pointer to this Type
        setfinalcall.getArgument(0).getType().(PointerType).getBaseType() = this
        and
        // The second argument of the SetFinalizer call is our 'final' function
        setfinalcall.getArgument(1) = final.getARead().asExpr()
        and
        // The 'final' function includes a call to a 'freeing' function
```

```
        final.getFuncDecl().getAChild*() = freecall
        and
        // WARNING: The definition of a freeing function is hardcoded by name
        // if other freeing functions are ever used it will not be detected
        (freecall.getCalleeName().matches("%free") or
freecall.getCalleeName().matches("%cleanup"))
        and
        // The variable v is a pointer to this Type
        v.getType().(PointerType).getBaseType() = this
        and
        // The initial read is of the form field_base.finalfield
        init_read.readsField(field_base,finalfield)
        and
        // The field base is actually the variable v
        v.getARead() = field_base
        and
        // Data flows from the initial read to the free read
        init_read.getASuccessor*() = free_read
        and
        // The freecall includes the free read as one of its arguments
        freecall.getAnArgument().getAChildExpr*() = free_read.asExpr()
    }

    Field getFinalField() {
        result = finalfield
    }

    CallExpr getFreeCall() {
        result = freecall
    }
}

/** A runtime.KeepAlive() call where one of the arguments is our target Type */
class RuntimeKeepAliveCall extends CallExpr {
    BoringTypeWithCF bt;
    Expr arg;

    RuntimeKeepAliveCall() {
        // The target function of this call is from the runtime package
        this.getTarget().getPackage().getName() = "runtime"
        and
        // The target function is called KeepAlive
        this.getTarget().getName() = "KeepAlive"
        and
        // One of the arguments is of our target Type
        this.getAnArgument() = arg
```

```
        and
        arg.getAChildExpr*().getType().(PointerType).getBaseType() = bt
    }

    Expr getArg() {
        result = arg
    }
}

/** A deferred runtime.KeepAlive() call where one of the arguments is our target Type
*/
class DeferredKeepAlive extends DeferStmt {
    RuntimeKeepAliveCall rkac;
    DeferredKeepAlive(){
        // The defer statement includes a RuntimeKeepAliveCall
        this.getAChild*() = rkac
    }

    RuntimeKeepAliveCall getRKAC() {
        result = rkac
    }

    Expr getArg() {
        result = rkac.getArg()
    }
}

/** A read of a variable of our target Type that implicitly keeps it alive */
class ImplicitKeepAliveRead extends DataFlow::ReadNode {
    BoringTypeWithCF bt;
    Variable v;

    ImplicitKeepAliveRead() {
        // The variable v is a pointer to our target Type
        v.getType().(PointerType).getBaseType() = bt
        and
        // This is a read of v
        this.reads(v)
    }

    Variable getReadVar() {
        result = v
    }
}

/**
```

```
 * A read that accesses the field of our Type which is freed
 * by the custom finalizer. We are interested when this happens
 * in a function call.
 * */
class ReadFieldDanger extends DataFlow::ReadNode {
    BoringTypeWithCF bt;
    DataFlow::Node field_base;

    ReadFieldDanger() {
        // This reads field_base.field, where field is something that gets freed
        // or cleaned by the custom finalizer of our target Type
        this.readsField(field_base,bt.getFinalField())
        and
        // field_base is a pointer to our target Type
        field_base.asExpr().getType().(PointerType).getBaseType() = bt
    }

    BoringTypeWithCF getBT() {
        result = bt
    }

    DataFlow::Node getFB() {
        result = field_base
    }

}

/**
 * A dangerous read that is inside a function call.
 * We exclude calls to the 'freeing' functions inside
 * the custom finalizer.
 * */
class CallDanger extends DataFlow::CallNode {
    CallExpr call;
    ReadFieldDanger rfd;
    CallDanger() {
        // This CallNode corresponds to a CallExpr
        this.asExpr() = call
        and
        // The dangerous read is one of the function arguments
        call.getAnArgument().getAChildExpr*() = rfd.asExpr()
        and
        // The dangerous read is not the 'freeing' function
        not rfd.getBT().getFreeCall() = call
    }
```

```
    ReadFieldDanger getRFD() {
        result = rfd
    }
}

/** A dangerous call that is protected by a defer statement */
class ProtectedByDefer extends CallDanger {
    ControlFlow::Node n_call;
    ControlFlow::Node n_dka;
    DeferredKeepAlive dka;
    Variable v;

    ProtectedByDefer() {
        // We need to convert the DataFlow node to a ControlFlow node
        n_call.isFirstNodeOf(this.asExpr())
        and
        // In the CFG there is a preceding node with a defer statement
        // that dominates the node of our call
        n_dka.isFirstNodeOf(dka)
        and
        n_dka.dominatesNode(n_call)
        and
        // The defer statement reads the same variable as the dangerous read
        dka.getArg().getAChildExpr*() = v.getARead().asExpr()
        and
        this.getRFD().getFB() = v.getARead()
    }
}

/** A dangerous call that is protected by a runtime.KeepAlive() call */
class ProtectedByRKA extends CallDanger {
    ControlFlow::Node n;
    RuntimeKeepAliveCall rkac;
    Variable v;

    ProtectedByRKA() {
        // We need to convert the DataFlow node to a ControlFlow node
        n.isFirstNodeOf(this.asExpr())
        and
        // In the CFG there is a successor node with a KeepAlive() call
        // We should use postDominates here, but it is not available for Go?
        n.getASuccessor*().isFirstNodeOf(rkac)
        and
        // The KeepAlive() call reads the same variable as the dangerous read
        rkac.getArg().getAChildExpr*() = v.getARead().asExpr()
        and
```

```
        this.getRFD().getFB() = v.getARead()
    }
}

/** A dangerous call that is protected by a later read of the variable */
class ProtectedByIKAR extends CallDanger {
    ControlFlow::Node n;
    ImplicitKeepAliveRead ikar;

    ProtectedByIKAR() {
        n.isFirstNodeOf(this.asExpr())
        and
        n.getASuccessor+().isFirstNodeOf(ikar.asExpr())
        and
        not this.getRFD().getFB() = ikar
        and
        ikar.getReadVar().getARead() = this.getRFD().getFB()
    }

    ImplicitKeepAliveRead getProtectingIKAR() {
        result = ikar
    }
}

//We select all dangerous calls that are not protected in some way
from CallDanger call_danger
where
not call_danger instanceof ProtectedByDefer
and
not call_danger instanceof ProtectedByRKA
and
not call_danger instanceof ProtectedByIKAR
  select call_danger.getRFD().getFB().asExpr(), "This object may be collected at the
    start of this function call, triggering its custom finalizer that will free the
                                    accessed field."
```

Figure C.3: CodeQL query that detects TOB-GOCL-3

D. Code Quality Findings

We identified the following code quality issue through manual and automatic code review.

- **Unnecessary field in `boring` ECDH public key.** The `PublicKeyECDH` type of the `boring` package has a `group` field that is not used. We recommend removing this field.

E. Fix Review Results

When undertaking a fix review, Trail of Bits reviews the fixes implemented for issues identified in the original report. This work involves a review of specific areas of the source code and system configuration, not comprehensive analysis of the system.

On March 24, 2025, Trail of Bits reviewed the fixes and mitigations implemented by the Go team for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue.

In summary, Google has resolved all six issues identified in this report. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Severity	Status
1	Fiat conversion from bytes to field elements is not constant time	Informational	Resolved
2	P-256 conditional negation is not constant time in PowerPC assembly	Informational	Resolved
3	Custom finalizer may free memory at the start of a C function call using this memory	Low	Resolved
4	The CTR-DRBG module presents multiple misuse risks	Informational	Resolved
5	PBKDF2 does not enforce output length limitations	Informational	Resolved
6	Timing Leak in edwards25519 Scalar.SetCanonicalBytes	Informational	Resolved

Detailed Fix Review Results

TOB-GOCL-1: Fiat conversion from bytes to field elements is not constant time

Resolved. The `subtle` module has been updated with a new function, `ConstantTimeLessOrEqBytes`, which performs the required comparison in constant time. The comparisons in each FIPS curve module have been replaced with calls to the new function.

TOB-GOCL-2: P-256 conditional negation is not constant time in PowerPC assembly

Resolved. The conditional early exit has been replaced with a constant-time conditional move at the end of the function, rendering it constant-time.

TOB-GOCL-3: Custom finalizer may free memory at the start of a C function call using this memory

Resolved. Appropriate deferred calls to runtime.KeepAlive have been added to the ECDH function.

TOB-GOCL-4: The CTR-DRBG module presents multiple misuse risks

Resolved. Documentation has been added to the CTR-DRBG code to indicate the scoping and functionality of the CTR-DRBG implementation and to warn against its use elsewhere.

TOB-GOCL-5: PBKDF2 does not enforce output length limitations

Resolved. The internal implementation of PBKDF2 has been updated to enforce the output length. Documentation has been added to note the limitation, and tests have been added to ensure that the limitations are enforced and that overflowing request lengths do not cause problems.

TOB-GOCL-6: Timing Leak in edwards25519 Scalar.SetCanonicalBytes

Resolved. The `isReduced` function has been updated to use a constant-time subtraction technique instead of a loop with a conditional return.

Code Quality Findings

Resolved. The `group` field of the `PublicKeyECDH` struct has been removed.

F. Fix Review Status Categories

The following table describes the statuses used to indicate whether an issue has been sufficiently addressed.

Fix Status	
Status	**Description**
Undetermined	The status of the issue was not determined during this engagement.
Unresolved	The issue persists and has not been resolved.
Partially Resolved	The issue persists but has been partially resolved.
Resolved	The issue has been sufficiently resolved.

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test the controls and security properties of software. These techniques augment our manual security review work, but each has its limitations: for example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. Their use is also limited by the time and resource constraints of a project.