Bron MPC

Security Assessment

January 8, 2026

Prepared for:
Alireza Rafiei
Bron Labs

Prepared by: **Opal Wright, Fredrik Dahlgren, Keegan Ryan, and Joop van de Pol**

Table of Contents

Project Summary

Contact Information

The following project manager was associated with this project:

Kimberly Espinoza, Project Manager
kimberly.espinoza@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:

Opal Wright, Consultant
opal.wright@trailofbits.com

Fredrik Dahlgren, Consultant
fredrik.dahlgren@trailofbits.com

Keegan Ryan, Consultant
keegan.ryan@trailofbits.com

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

Project Timeline

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

Date	Event
October 15, 2025	Pre-project kickoff call
October 28, 2025	Status update meeting #1
November 3, 2025	Status update meeting #2
November 10, 2025	Status update meeting #3
November 19, 2025	Delivery of report draft and readout meeting
December 19, 2025	Completion of fix review
January 8, 2026	Delivery of final comprehensive report with fix review

Executive Summary

Engagement Overview

Bron Labs engaged Trail of Bits to review the security of the Bron MPC library. The review was scoped to cover Bron Labs implementations of the threshold signature protocols DKLs23, Lindell22, Lindell17, and Boldyreva03; the distributed key generation protocol GJKR07; and the implementations of HPKE, CTR-DRBG, POSEIDON, and KMAC.

The library implements several threshold signature schemes, including the DKLs23 and Lindell17 schemes for ECDSA, Lindell22 for Schnorr signatures, and Boldyreva03 for BLS signatures. It further implements the underlying dependencies of these schemes, including distributed key generation (DKG), elliptic curve arithmetic for several curves, HPKE, CTR-DRBG from NIST 800-90A, Poseidon, KMAC, Oblivious Transfer, Paillier Encryption, hash and Pedersen commitments, Shamir Secret Sharing with Feldman verification, and all corresponding zero-knowledge proofs based on the Fiat-Shamir and Fischlin transforms.

A team of four consultants conducted the review from October 20 to November 17, 2025, for a total of eight engineer-weeks of effort. Our testing efforts focused on the different threshold signature scheme implementations and their dependencies. With full access to source code and documentation, we performed static and dynamic testing of the Bron MPC library, using automated and manual processes. From December 17 to December 19, 2025, one consultant performed a review of the fixes provided by Bron Labs, as described in appendix G.

As part of this project, we also reviewed the library for issues related to misuse resistance and denial of service (DoS). However, Bron Labs considers DoS attacks out of scope for its library due to its intended usage. All findings related to such attacks have been classified as low severity as a result, except where they enable other attacks. Furthermore, the current version of the library does not fully implement identifiable abort. Specifically, a user cannot always identify parties mounting selective abort attacks on the DKLs23 protocol. When users see a predetermined number of errors related to those attacks, they must stop using the key pending manual review. In addition to this, the library also contains a number of experimental features, which were not in scope.

Observations and Impact

We identified several missing checks in the various protocols, leading to a key recovery attack on Lindell17 (TOB-BRON-23) and a rogue-key attack on the Gennaro DKG (TOB-BRON-13). In addition, the library supports users with a zero ID, and provides a convenience function that would generate such users, leading to key exposure (TOB-BRON-11, TOB-BRON-16). We further identified deviations from the specification that cannot clearly be exploited but invalidate the corresponding security proofs (TOB-BRON-2, TOB-BRON-3, TOB-BRON-5, TOB-BRON-6, TOB-BRON-8, TOB-BRON-14). Several zero-knowledge primitives have implementation flaws leading to proof forgery (TOB-BRON-15, TOB-BRON-18, TOB-BRON-19) and proof replay (TOB-BRON-9).

Recommendations

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

- **Remediate the findings disclosed in this report.** These findings should be addressed through direct fixes or broader refactoring efforts.
- **Update the specification.** The current specification of the various protocols is outdated. Because several of the protocols have been modified from the version presented in the original papers, it is important to capture and justify these modifications in a detailed specification. The specification should also capture assumptions that each protocol relies on (e.g., with respect to honest majority requirements).
- **Add negative testing.** Each protocol description includes several checks that participants must perform during the protocol. There should be negative tests (or property tests) covering each of these checks that ensure that all violations are caught and handled by the implementation. Furthermore, there should be tests for malicious participants that send invalid elements or collections of elements of incorrect lengths (empty, too short, too long).
- **Place experimental features in a dedicated package.** It is currently not clear which parts of the library are experimental. Placing experimental features in a dedicated package will help library users avoid them when appropriate.

Finding Severities and Categories

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

EXPOSURE ANALYSIS

Severity	*Count*
High	2
Medium	1
Low	6
Informational	12
Undetermined	4

CATEGORY BREAKDOWN

Category	*Count*
Cryptography	17
Data Validation	7
Testing	1

Project Goals

The engagement was scoped to provide a security assessment of the Bron Labs MPC library. Specifically, we sought to answer the following non-exhaustive list of questions:

- Do internal protocol specifications deviate from the corresponding paper or papers?
- Do implementations adhere to the corresponding internal specifications?
- Are underlying primitives like zero-knowledge proofs, verifiable secret sharing (VSS), HPKE, and KMAC implemented correctly?
- Are protocol artifacts like zero-knowledge proofs bound to the sender and protocol instance to prevent replay attacks?
- Are untrusted inputs validated correctly by all implementation components?
- Are security parameters chosen conservatively and consistently across the library?
- Are random values sampled securely from the correct ranges?
- Is the unit and integration testing coverage sufficient to ensure security against malicious or malformed inputs?
- Are project dependencies updated in a timely manner to avoid introducing vulnerabilities in third-party code into the codebase?

Project Targets

The engagement involved reviewing and testing the following target. The initial stages of the review focused on the commits `84f1d9031d1fd63838f1e1fc250cfc2bb550050c` and `e01154cb34bda067fb4b1365b964afbcc554e6d5`. The findings in this report reference the commits where the issues were initially identified.

Bron MPC

Repository	https://github.com/bronlabs/bron-crypto
Version	`4dfa2c76a4d7661372add12511de202e4c19c9e4`

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:

- **Specification review.** We reviewed each specification against the corresponding academic paper or papers to ensure that the specification matches the protocol described by the latest versions of the referenced papers. We also reviewed all changes introduced to ensure that they did not compromise the security properties of the high-level protocol.
- **Manual code review.** We manually reviewed each protocol implementation against the corresponding specification and referenced papers. Apart from general correctness and robustness issues, we also looked for issues related to weak input validation, insufficient misuse resistance, and error handling.
- **Static analysis.** We ran Semgrep with an automatically configured ruleset, Trail of Bits public rulesets, and a Trail of Bits internal repository of rules. We ran CodeQL with the default Go query suites, and Trail of Bits public query suites for Go. We also ran the `golangci-lint` linter.
- **Dependency analysis.** We ran Nancy to identify project dependencies with known vulnerabilities.
- **Test-coverage analysis.** We used the Go `cover` tool to identify parts of the codebase with insufficient test coverage.

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:

- **Resistance against selective abort attacks.** These were specifically ruled out of scope for the review by Bron. For the DKLs23 scheme, this includes attacks using selective aborts on the OT protocols, but also those related to any of the other consistency checks in the DKLs23 protocol. For Lindell17, these include broken record attacks. The library user will need to handle errors from the library appropriately and a failure to do so may lead to private key exposure. In general, the current error reporting mechanisms of the library increase the attack surface for denial-of-service attacks, because users cannot sufficiently distinguish certain critical errors.
- **Resistance against timing attacks.** Bron has chosen specific parts of the library and implemented those parts using constant-time code on a best-effort basis. Other

parts of the library were deliberately not hardened against timing attacks. While some potential issues were identified in the hardened implementations (see appendix D), it is impossible to assess whether the hardening is fully successful even where we did not identify any issues.

- **Lindell17 and Paillier implementation.** The limited review of the Lindell17 and underlying Paillier implementation found a relatively large number of issues compared to the other components. We recommend additional review once the implementation is more stable.
- **Third-party dependencies.** While this library implements many of the underlying dependencies of each protocol, it still relies on a number of third-party libraries (such as `saferith`). These have not been reviewed.

Automated Testing

Trail of Bits uses automated techniques to extensively test the security properties of software. We use both open-source static analysis and fuzzing utilities, along with tools developed in house, to perform automated testing of source code and compiled software.

Test Harness Configuration

We used the following tools in the automated testing phase of this project. For information on how to configure and run each tool, see appendix C.

Tool	Description
CodeQL	A code analysis engine developed by GitHub to automate security checks
`go-mod-outdated`	A tool that identifies outdated Go dependencies
`go-test`	A subcommand for the Go compiler that can be used to generate test coverage data for the codebase
`golangci-lint`	A meta-linter for Go that runs multiple linters and static analysis tools on the codebase and aggregates the results
Nancy	A tool to check for known vulnerabilities in Go dependencies
Semgrep	An open-source static analysis tool for finding bugs and enforcing code standards when editing or committing code and during build time

Areas of Focus

Our automated testing and verification work focused on the following questions:

- Does the codebase contain commonly known vulnerabilities or weaknesses, as identified by `golangci-lint`, CodeQL, and Semgrep?
- Does the codebase have any outdated dependencies or dependencies with known vulnerabilities?
- Is the unit and integration test coverage across the codebase sufficient?

Test Results

The results of this focused testing are detailed below.

Known vulnerable code patterns: To identify unidiomatic code patterns and known vulnerabilities, we ran `golangci-lint`, CodeQL, and Semgrep on the Bron Labs codebase. We also wrote a custom Semgrep rule to perform variant analysis on one of the issues identified during the engagement.

Property	Tool	Result
Does the codebase contain unidiomatic code patterns or weaknesses identified by code linters?	`golangci-lint`	Appendix C
Does the codebase contain known vulnerable code patterns identified by static analysis tools?	CodeQL, Semgrep	TOB-BRON-20

Dependency management: We used Nancy and `go-mod-outdated` to identify outdated project dependencies and dependencies with known vulnerabilities.

Property	Tool	Result
Does the project have outdated dependencies?	`go-mod-outdated`	**Passed**
Does the project have dependencies with known vulnerabilities that affect the security of the codebase?	Nancy	**Passed**

Unit and integration test coverage: We used the Go compiler and the Go cover tool to review the unit and integration test coverage across the codebase.

Property	Tool	Result
Is the unit and integration test coverage sufficient?	`go-test`	TOB-BRON-25

Summary of Findings

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

ID	Title	Type	Severity
1	Signature aggregation rejects valid recovery IDs	Data Validation	**Informational**
2	The base OT receivers use the same choice bits for all senders in DKLs23	Cryptography	**Undetermined**
3	VSOT with naive batching does not provide endemic security	Cryptography	**Undetermined**
4	VSOT sender inconsistently rejects identity point	Cryptography	**Informational**
5	SoftSpokenOT does not include all row indices in the final hash	Cryptography	**Undetermined**
6	OT extension PRG is not necessarily unique due to potential seed collision	Cryptography	**Informational**
7	BBOT output conversion can include additional domain separation	Cryptography	**Informational**
8	DKLs23 consistency check failures have wrong error type	Cryptography	**Undetermined**
9	Fischlin proofs are not bound to the prover	Cryptography	**Low**
10	The Fischlin verifier panics on malicious inputs	Data Validation	**Low**
11	The library does not reject zero sharing IDs	Data Validation	**Informational**
12	Malicious participant can increase sharing threshold for Feldman and Pedersen VSS	Cryptography	**Medium**

13	Gennaro DKG is vulnerable to rogue-key attacks by a malicious participant	Cryptography	**High**
14	Paillier nonces are sampled from the wrong space	Cryptography	**Informational**
15	The LPDL verifier accepts forged proofs	Cryptography	**Low**
16	The library generates zero-sharing IDs	Cryptography	**Informational**
17	BLS key generation does not match standard recommendations	Cryptography	**Informational**
18	The pailliern proof does not incorporate statement	Cryptography	**Informational**
19	Paillier range proof serialization problems	Cryptography	**Low**
20	Public key material equality test does not ensure equality	Data Validation	**Informational**
21	Inconsistent Paillier plaintext representations	Data Validation	**Informational**
22	Ineffective input validation in LPDL verifier	Data Validation	**Informational**
23	Malicious Alice can use invalid Paillier ciphertext to recover Lindell17 key	Cryptography	**High**
24	Paillier range proof verifier fails to check input lengths	Data Validation	**Low**
25	Insufficient test coverage	Testing	**Low**

Detailed Findings

1. Signature aggregation rejects valid recovery IDs	
Severity: **Informational**	Difficulty: **N/A**
Type: Data Validation	Finding ID: TOB-BRON-1
Target: `pkg/signatures/ecdsa/signature.go`, `pkg/signatures/ecdsa/ecdsa.go`	

Description

Signature aggregation rejects recovery IDs larger than a one-bit value, but an ECDSA recovery ID comprises two bits. Specifically, the `NewSignature` function rejects values that are not zero or one.

```
if v != nil && (*v != 0 && *v != 1) {
	return nil, errs.NewFailed("v must be 0 or 1")
}
```

Figure 1.1: This check rejects any recovery ID that is not zero or one. (`bron-crypto/pkg/signatures/ecdsa/signature.go#L21–L23`)

Conversely, constructing a recovery ID using `ComputeRecoveryId` supports values that are two or three in the case that the x-coordinate `rx` is larger than the group modulus.

```
var recoveryId int
if !ry.IsOdd() {
	recoveryId = 0
} else {
	recoveryId = 1
}

if base.PartialCompare(rx.Cardinal(),
subGroupOrder).Is(base.Ordering(base.GreaterThan)) {
	recoveryId += 2
}
```

Figure 1.2: The recovery ID computation can produce a recovery ID ranging from zero to three. (`bron-crypto/pkg/signatures/ecdsa/ecdsa.go#L50–L59`)

Recommendations

Short term, add support for new signatures with recovery ID equal to two or three.

Long term, add corresponding tests for curves where this happens with nonnegligible probability.

2. The base OT receivers use the same choice bits for all senders in DKLs23

Severity: **Undetermined**	Difficulty: **Undetermined**
Type: Cryptography	Finding ID: TOB-BRON-2
Target: `pkg/threshold/tsig/tecdsa/dkls23/signing/interactive/sign/rounds.go`, `pkg/threshold/tsig/tecdsa/dkls23/keygen/dkg/rounds.go`	

Description
When performing base OT as a receiver in DKLs23, each participant generates one set of choice bits and reuses this set for each of the corresponding senders. The following code fragment shows this for the DKLs23 variant that includes base OTs as part of signing.

```
choices := make([]byte, (softspoken.Kappa+7)/8)
_, err = io.ReadFull(c.prng, choices)
if err != nil {
	return nil, errs.WrapRandomSample(err, "cannot sample choices")
}

r2u := hashmap.NewComparable[sharing.ID, *Round2P2P[P, B, S]]()
for id, u := range outgoingP2PMessages(c, r2u) {
	otR1u, ok := otR1.Get(id)
	if !ok {
		return nil, errs.NewFailed("cannot run round 2 of VSOT setup party")
	}
	var seed *ecbbot.ReceiverOutput[S]
	u.OtR2, seed, err = c.baseOtReceivers[id].Round2(otR1u, choices)
	if err != nil {
		return nil, errs.WrapFailed(err, "cannot run round 2 of VSOT party")
	}
	c.state.baseOtReceiverOutputs[id], err =
seed.ToBitsOutput(baseOtMessageLength)
	if err != nil {
		return nil, errs.WrapFailed(err, "cannot convert seed to bits output")
	}
}
```

Figure 2.1: The random choices are reused for all senders in the second round function of the DKLs23 signing protocol.
(`bron-crypto/pkg/threshold/tsig/tecdsa/dkls23/signing/interactive/sign/rounds.go#L50–L71`)

Note that the DKG of another DKLs23 variant has the same issue.

```
choices := make([]byte, (softspoken.Kappa+7)/8)
_, err = io.ReadFull(p.prng, choices)
if err != nil {
	return nil, nil, errs.WrapRandomSample(err, "cannot sample choices")
}

p.state.receiverSeeds = hashmap.NewComparable[sharing.ID, *vsot.ReceiverOutput]()
r2b := &Round2Broadcast[P, B, S]{
	GennaroR2: gennaroR2b,
}
r2u := hashmap.NewComparable[sharing.ID, *Round2P2P[P, B, S]]()
for id, u := range outgoingP2PMessages(p, r2u) {
	var ok bool
	var err error
	var seed *vsot.ReceiverOutput

	u.GennaroR2, ok = gennaroR2u.Get(id)
	if !ok {
		return nil, nil, errs.NewFailed("cannot run round 2 of Gennaro DKG 
party")
	}
	u.ZeroR2, ok = zeroR2u.Get(id)
	if !ok {
		return nil, nil, errs.NewFailed("cannot run round 2 of PRZS setup 
party")
	}
	otR1u, ok := otR1.Get(id)
	if !ok {
		return nil, nil, errs.NewFailed("cannot run round 2 of VSOT setup 
party")
	}
	u.OtR2, seed, err = p.baseOTReceivers[id].Round2(otR1u, choices)
	if err != nil {
		return nil, nil, errs.WrapFailed(err, "cannot run round 2 of VSOT 
party")
	}
	p.state.receiverSeeds.Put(id, seed)
}
```

Figure 2.2: The random choices are reused for all senders in the second round function of the DKLs23 DKG protocol.
(bron-crypto/pkg/threshold/tsig/tecdsa/dkls23/keygen/dkg/rounds.go#L63–L96)

It is not immediately clear whether this can be exploited, because each base OT and OT extension instance includes domain separation using participant indices to ensure that, even though honest receivers may reuse choice bits for the base OT, the resulting OT output strings are not correlated between different instances.

That said, this violates the specifications of the involved primitives and it is unclear whether the corresponding security proofs apply in this setting.

Recommendations

Short term, move the choice generation inside the loop in the second round functions for the DKLs23 signing and DKG protocols to ensure that each base OT instance has independent choice bits.

Long term, add negative tests that verify important invariants in the specification, such as independence of specific randomly generated values and efficacy of domain separation.

3. VSOT with naive batching does not provide endemic security

Severity: **Undetermined**	Difficulty: **Undetermined**
Type: Cryptography	Finding ID: TOB-BRON-3
Target: `pkg/ot/base/vsot/rounds.go`	

Description

The VSOT implementation relies on naive batching, and therefore does not achieve endemic security as described in lemma 1 of the batching OT paper.

```
for i := range r.suite.Xi() {
	c := uint64((choices[i/8] >> (i % 8)) & 0b1)
	receiverOutput.Messages[i] = make([][]byte, r.suite.L())
	for j := range r.suite.L() {
		idx := i*r.suite.L() + j
		a, err := r.suite.field.Random(r.prng)
		if err != nil {
			return nil, nil, errs.WrapRandomSample(err, "generating a")
		}
		r.state.omegaRaw[idx] = c
		r.state.omega[idx] = r.suite.field.FromUint64(r.state.omegaRaw[idx])
		r.state.bigA[idx] =
r.suite.curve.ScalarBaseMul(a).Add(r.state.bigB.ScalarMul(r.state.omega[idx]))
		r.state.rhoOmega[idx], err = r.hash(r.state.bigB, r.state.bigA[idx],
r.state.bigB.ScalarMul(a).ToCompressed())
		if err != nil {
			return nil, nil, errs.WrapHashing(err, "cannot hash B * a_i")
		}
		receiverOutput.Messages[i][j] = r.state.rhoOmega[idx]

		r.tape.AppendBytes(fmt.Sprintf("%s%d-%d-", aLabel, i, j),
r.state.bigA[idx].ToCompressed())
	}
}
```

Figure 3.1: The hash function does not include the index.
(`bron-crypto/pkg/ot/base/vsot/rounds.go#L103–L123`)

The base VSOT implementation extends the implementation from appendix A of the DKLs18 paper, which claims this does not sacrifice security. This extension includes hashing both public keys `(B, A_idx)` in each iteration `idx`. However, this still does not protect against a malicious receiver that sends the same `A_idx = A` in every iteration. The resulting OT output pairs `(rho_0, rho_1)` will have the same value regardless of batch index. This violates the endemic OT security notion that states that a malicious receiver can choose one of the output values, but the other one is chosen uniformly at random for each batch index.

We did not identify any way to directly exploit this, as the library implements only SoftSpokenOT as the OT extension protocol (based on KOS15). The batching OT paper states the following:

> *We are not sure whether a more sophisticated attack on the base OTs (even for a specific naively batched OT) can break KOS OT extension.*

Recommendations

Short term, add the index `idx` to the hash function when deriving pads.

Long term, add negative tests for some malicious receiver scenarios to ensure that the resulting pads not chosen by the receiver are random.

4. VSOT sender inconsistently rejects identity point

Severity: **Informational**	Difficulty: **N/A**
Type: Cryptography	Finding ID: TOB-BRON-4
Target: `pkg/ot/base/vsot/rounds.go`	

Description

When computing the two pads `rho_0` and `rho_1`, the VSOT sender ensures that the first is not calculated using the elliptic curve identity point, but fails to ensure the same for the second pad. A malicious receiver can send `A = B`, which causes the sender to process `A - B = 0`, as shown in the following code fragment.

```
for i := range s.suite.Xi() {
	senderOutput.Messages[i][0] = make([][]byte, s.suite.L())
	senderOutput.Messages[i][1] = make([][]byte, s.suite.L())
	for j := range s.suite.L() {
		idx := i*s.suite.L() + j
		bigA := r2.BigA[idx]
		if bigA.IsOpIdentity() {
			return nil, nil, errs.NewValidation("A is identity")
		}

		rho0[idx], err = s.hash(s.state.bigB, bigA,
bigA.ScalarMul(s.state.b).ToCompressed())
		if err != nil {
			return nil, nil, errs.WrapHashing(err, "cannot hash A * b_i")
		}
		rho1[idx], err = s.hash(s.state.bigB, bigA,
(bigA.Sub(s.state.bigB)).ScalarMul(s.state.b).ToCompressed())
		if err != nil {
			return nil, nil, errs.WrapHashing(err, "cannot hash (A - B_i) *
b_i")
		}
		senderOutput.Messages[i][0][j] = rho0[idx]
		senderOutput.Messages[i][1][j] = rho1[idx]
```

Figure 4.1: The sender rejects the identity point, but not its own public key.
(`bron-crypto/pkg/ot/base/vsot/rounds.go#L152–L171`)

We did not discover any way to exploit this, because the underlying scalar multiplication routines properly handle the point at infinity. Sending `A = B` is equivalent to the receiver using the random scalar `a = 0` and choice bit `omega = 1`. This would leak their choice bit to the sender, because an honest receiver cannot send `A = B` for choice bit `omega = 0` unless `a = b`, which implies that the receiver can solve the discrete logarithm problem for the public key `B` of the sender. However, it seems inconsistent to check for `A = 0` but not to check for `A - B = 0`.

Recommendations

Short term, either add a check that `A = B` or remove the check that `A = 0`.

Long term, update the specification to match the performed checks and consider adding a justification for why checks are performed.

5. SoftSpokenOT does not include all row indices in the final hash

Severity: **Undetermined**	Difficulty: **Undetermined**
Type: Cryptography	Finding ID: TOB-BRON-5
Target: `pkg/ot/extension/softspoken/rounds.go`	

Description

The SoftSpokenOT implementation does not include all row indices in the final hash, as it includes the index `j` of the current batch but omits the index `l` of the rows within the current batch.

```
for j := 0; j < s.suite.Xi(); j++ {
	senderOutput.Messages[j][0] = make([][]byte, s.suite.L())
	senderOutput.Messages[j][1] = make([][]byte, s.suite.L())
	for l := 0; l < s.suite.L(); l++ {
		digest, err := hashing.Hash(sha256.New, s.sessionId[:],
binary.LittleEndian.AppendUint32(nil, uint32(j)), qjTransposed[j*s.suite.L()+l])
		if err != nil {
			return nil, errs.WrapHashing(err, "bad hashing q_j for
SoftSpoken COTe (T&R.2)")
		}
		senderOutput.Messages[j][0][l] = digest
		digest, err = hashing.Hash(sha256.New, s.sessionId[:],
binary.LittleEndian.AppendUint32(nil, uint32(j)),
qjTransposedPlusDelta[j*s.suite.L()+l])
		if err != nil {
			return nil, errs.WrapHashing(err, "bad hashing q_j_pDelta for
SoftSpoken COTe (T&R.2)")
		}
		senderOutput.Messages[j][1][l] = digest
	}
}
```

Figure 5.1: The hash function includes batch index `j`, but not row index `l`.
(`bron-crypto/pkg/ot/extension/softspoken/rounds.go#L185–L200`)

This is not in line with the SoftSpokenOT and KOS15 papers, which include the index `j` of each row (these papers do not consider batching). While we did not identify any attack that exploits this omission, the security proofs no longer hold.

Recommendations

Short term, add the index `l` to the hash input when decorrelating the rows.

Long term, include a mapping in your specification that shows how different variables and indices map to their corresponding counterparts in the academic references of the specified algorithms.

6. OT extension PRG is not necessarily unique due to potential seed collision

Severity: **Informational**	Difficulty: **N/A**
Type: Cryptography	Finding ID: TOB-BRON-6
Target: `pkg/ot/extension/softspoken/rounds.go`	

Description

The implementation of the SoftSpokenOT OT extension uses `ChaCha8` as its PRG, where the seed is the exclusive or of the session ID and the base OT message. While this ensures that the PRG will never repeat for the same base OT messages as long as the session ID is different, it does mean that there could be potentially colliding PRG streams between different sessions with different base OT messages.

```
var prngSeed [32]byte
subtle.XORBytes(prngSeed[:], r.senderSeeds.Messages[i][0][0], r.sessionId[:])
prng := rand.NewChaCha8(prngSeed)
if _, err = io.ReadFull(prng, t[0][i]); err != nil {
        return nil, nil, errs.WrapFailed(err, "bad PRG reading for SoftSpoken OTe")
}
subtle.XORBytes(prngSeed[:], r.senderSeeds.Messages[i][1][0], r.sessionId[:])
prng = rand.NewChaCha8(prngSeed)
if _, err = io.ReadFull(prng, t[1][i]); err != nil {
        return nil, nil, errs.WrapFailed(err, "bad PRG for SoftSpoken OTe")
}
```

Figure 6.1: The session ID and the base OT messages are combined to derive ChaCha8 seeds. (`bron-crypto/pkg/ot/extension/softspoken/rounds.go#L50–L60`)

Considering that endemic security for the base OTs implies that malicious participants can choose their output messages, they could deliberately cause such collisions, which would not be possible for an ideal PRG that is unique per session. However, we do not see how this is exploitable in practice.

Recommendations

Short term, replace `ChaCha8` with an extensible output function.

Long term, avoid using the exclusive or operation when handling session identifiers for the purposes of domain separation.

7. BBOT output conversion can include additional domain separation

Severity: **Informational**	Difficulty: **N/A**
Type: Cryptography	Finding ID: TOB-BRON-7
Target: `pkg/ot/base/ecbbot/bbot.go`	

Description

During DKLs23 signing, all participants convert the prime field elements from the BBOT output messages to byte arrays by using an unkeyed `Blake2b` hash. Changing this to a keyed hash (e.g., by using the session ID or some other transcript-derived value as a key) improves the domain separation by ensuring that the same base OT messages will map to different bytes in different sessions.

```
func (s *SenderOutput[S]) ToBitsOutput(b int) (*vsot.SenderOutput, error) {
	if b < 16 {
		return nil, errs.NewValidation("invalid hash size")
	}
	h, err := blake2b.New(b, nil)
	if err != nil {
		return nil, errs.NewFailed("failed to create hasher")
	}

	out := &vsot.SenderOutput{
		SenderOutput: ot.SenderOutput[[]byte]{
			Messages: make([][2][][]byte, len(s.Messages)),
		},
	}
	for xi := range s.Messages {
		out.Messages[xi][0] = make([][]byte, len(s.Messages[xi][0]))
		out.Messages[xi][1] = make([][]byte, len(s.Messages[xi][1]))
		for l := range s.Messages[xi][0] {
			h.Reset()
			h.Write(s.Messages[xi][0][l].Bytes())
			out.Messages[xi][0][l] = h.Sum(nil)
			h.Reset()
			h.Write(s.Messages[xi][1][l].Bytes())
			out.Messages[xi][1][l] = h.Sum(nil)
		}
	}

	return out, nil
}
```

Figure 7.1: Sender output conversion uses an unkeyed `Blake2b` hash.
(`bron-crypto/pkg/ot/base/ecbbot/bbot.go#L111–L139`)

We did not find any way to exploit the fact that the hash is currently unkeyed, because the BBOT procedures themselves already include domain separation where possible. However, adding additional domain separation here should be seen as a defense-in-depth measure.

Recommendations

Short term, use the session ID or another transcript-derived value as a `Blake2b` hash key when converting the BBOT output messages.

Long term, use domain separation when hashing elements wherever possible.

8. DKLs23 consistency check failures have wrong error type

Severity: **Undetermined**	Difficulty: **Undetermined**
Type: Cryptography	Finding ID: TOB-BRON-8

Target:
`pkg/threshold/tsig/tecdsa/dkls23/signing/interactive/{sign/rounds.go, sign_bbot/rounds.go, sign_softspoken/rounds.go}`

Description

The implementation returns a `Failed` error for a failure of the DKLs23 consistency checks, whereas this is supposed to be a `TotalAbort` or `IdentifiableAbort` according to the paper. Step 8 of the DKLs23 protocol requires participants to abort when the consistency check fails for any counterparty, and to refuse to participate in signing sessions with that counterparty in the future. The current error type does not reflect this for any of the three implementations of DKLs23.

```
if
!c.state.bigR[id].ScalarMul(c.state.chi[id]).Sub(message.p2p.GammaU).Equal(c.suite.C
urve().ScalarBaseMul(d[0])) {
	return nil, errs.NewFailed("consistency check failed")
}
if
!message.broadcast.Pk.ScalarMul(c.state.chi[id]).Sub(message.p2p.GammaV).Equal(c.sui
te.Curve().ScalarBaseMul(d[1])) {
	return nil, errs.NewFailed("consistency check failed")
}
```

Figure 25.1: *`bron-crypto/pkg/threshold/tsig/tecdsa/dkls23/signing/interactive/sign/rounds.go#L242–L247`*

It is not clear whether this is exploitable. The DKLs23 paper notes that a corrupt party that misbehaves would have a probability of at most 1/q to pass each consistency check where q is the curve order, but in general, the security proof does not seem to hold without these checks.

Recommendations

Short term, update the checks to generate `TotalAbort` or `IdentifiableAbort` errors.

Long term, add negative testing to ensure that each of the failures called out by the paper generate the correct error types.

9. Fischlin proofs are not bound to the prover

Severity: **Low**	Difficulty: **High**
Type: Cryptography	Finding ID: TOB-BRON-9
Target: `pkg/proofs/sigma/compiler/fischlin/prover.go`	

Description

Proofs generated by the Fischlin prover are cryptographically bound to the session by including the session ID in the `common-h` hash. However, they are not bound to the prover ID. This means that proofs could in principle be replayed by other participants within the same session.

```
func (p *prover[X, W, A, S, Z]) Prove(
    statement X, witness W) (compiler.NIZKPoKProof, error) {
    p.transcript.AppendBytes(
        rhoLabel, binary.LittleEndian.AppendUint64(nil, p.rho))
    p.transcript.AppendBytes(statementLabel, statement.Bytes())

    // [...]

    // 3. common-h ← H(x, m, sid)
    // (This is a full hash, with output length 2*κc)
    commonH, err := hashing.Hash(randomOracle, statement.Bytes(), a, p.sessionId[:])

    // [...]
    // 8. Output π
    return proofBytes, nil
}
```

Figure 9.1: The proof generated by the Fischlin compiler is not bound to the prover.
(`bron-crypto/pkg/proofs/sigma/compiler/fischlin/prover.go#L30–L121`)

If proofs are broadcast using a reliable broadcast protocol like echo broadcast, a malicious participant could replay a proof sent by another participant. If broadcast is implemented using a star-topology, the coordinator would also have to be compromised for malicious participants to be able to replay proofs.

Exploit Scenario

We do not know of an attack that exploits this issue to further compromise one of the higher-level protocols. However, since valid zero-knowledge proofs are required for simulatability of Lindell22, this breaks the security proof for that protocol.

Recommendations

Short term, include the sender’s ID as an input to the computation of the `common-h` hash.

Long term, use the transcript to compute `common-h` and include the sender ID in the transcript.

10. The Fischlin verifier panics on malicious inputs

Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-BRON-10
Target: `pkg/proofs/sigma/compiler/fischlin/verifier.go`	

Description

When the Fischlin proof verifier checks the proof `(A[i], E[i], Z[i])`, it copies the challenge `E[i]` into the `eBytes` byte array, which is then passed to the `Verify` function on the underlying Sigma protocol.

```
// [...]

// 4. For i ∈ {1, ..., ρ}
for i := uint64(0); i < fischlinProof.Rho; i++ {
	digest, err := v.hash(
		fischlinProof.B, commonH, i, fischlinProof.E[i], fischlinProof.Z[i])
	if err != nil {
		return errs.WrapHashing(err, "cannot compute digest")
	}

	// 4.b. Halt and output 'reject' if Hb(common-h, i, e_i, z_i) != 0
	if !isAllZeros(digest) {
		return errs.NewVerification("invalid challenge")
	}

	// 4.a. Halt and output 'reject' if VerifyProof(x, m_i, e_i, z_i) == 0
	eBytes := make([]byte, v.sigmaProtocol.GetChallengeBytesLength())
	copy(eBytes[len(eBytes)-len(fischlinProof.E[i]):], fischlinProof.E[i])
	err = v.sigmaProtocol.Verify(
		statement, fischlinProof.A[i], eBytes, fischlinProof.Z[i])
	if err != nil {
		return errs.WrapVerification(err, "verification failed")
	}
}

// [...]
```

Figure 10.1: A malicious prover can choose `E[i]` longer than `eBytes` to cause the verifier to panic. (`bron-crypto/pkg/proofs/sigma/compiler/fischlin/verifier.go#71–90`)

However, if a faulty or malicious prover sends a challenge `E[i]` that is longer than the challenge size of the Sigma protocol, then `len(eBytes) - len(fischlinProof.E[i])` will be negative and the call to copy will panic.

We wrote a Semgrep rule to find other variants of this issue within the codebase, but it did not identify any additional vulnerabilities.

Exploit Scenario
A malicious prover sends invalid zero-knowledge proofs to other participants, causing hard-to-diagnose panics in time-sensitive signing sessions.

Recommendations
Short term, check the size of the challenge `E[i]` before copying it to the `eBytes` byte array in the Fischlin verifier.

Long term, use property testing to strengthen input validation by exercising the codebase on unexpected and invalid inputs.

11. The library does not reject zero sharing IDs

Severity: **Informational**	Difficulty: **N/A**
Type: Data Validation	Finding ID: TOB-BRON-11
Target: `pkg/proofs/sigma/compiler/fischlin/verifier.go`	

Description

The library's public APIs for key generation accept zero sharing IDs, leaving validation of participant IDs to the end user. This type of validation is necessary to ensure that the shared secret is not exposed to one of the participants. However, it is not obvious to end users that it is up to them to perform this type of validation. To demonstrate the problem, we wrote a unit test that essentially copies the `Test_HappyPath` unit test for the DKLs23 DKG, but contains a small change to include a participant with a zero sharing ID.

```
// copy of the unit test Test_HappyPath
func Test_ShouldRejectZeroSharingId(t *testing.T) {
	t.Parallel()

	const THRESHOLD = 2
	const TOTAL = 3

	curve := k256.NewCurve()
	shareholders := hashset.NewComparable[sharing.ID]()
	for i := 0; i < TOTAL; i++ {
		shareholders.Add(sharing.ID(i)) // includes zero
	}
	accessStructure, err :=
		shamir.NewAccessStructure(THRESHOLD, shareholders.Freeze())
	require.NoError(t, err)

	// run DKLs23 DKG
	shards := testutils.RunDKLs23DKG(t, curve, accessStructure)
	require.Len(t, shards, TOTAL)

	// reconstruct secret
	shares := make([]*feldman.Share[*k256.Scalar], 0, len(shards))
	for _, shard := range shards {
		shares = append(shares, shard.Share())
	}
	feldmanScheme, err := feldman.NewScheme(
		curve.Generator(),
		accessStructure.Threshold(),
		accessStructure.Shareholders())
	secret, _ := feldmanScheme.Reconstruct(shares...)
```

```
        // verify that secret matches share with ID zero
        share := shards[sharing.ID(0)].Share()
        require.Equal(t, secret.Value(), share.Value())
}
```

Figure 11.1: Running the DKLs23 DKG with a zero sharing ID is not rejected, as shown by the unit test above.

The test passes, showing that zero sharing IDs are not rejected by either the access structure or the Feldman VSS implementations.

Recommendations

Short term, ensure that sharing IDs are nonzero (as scalar field elements) when the access structure is created. Alternatively, check that the ID is nonzero when the individual shares are generated in the `Scheme.DealAndRevealDealerFunc` function.

Long term, have the access structure determine the sharing IDs for the threshold and group size. For example, convert 32-bit unsigned participant IDs to 64-bit sharing IDs (to prevent overflows) and then add one, or generate the sharing ID by hashing the participant ID.

12. Malicious participant can increase sharing threshold for Feldman and Pedersen VSS

Severity: **Medium**	Difficulty: **Medium**
Type: Cryptography	Finding ID: TOB-BRON-12
Target: `pkg/threshold/dkg/gennaro/rounds.go`, `pkg/threshold/sharing/feldman/scheme.go`, `pkg/base/polynomials/polynomial_module.go`, `pkg/threshold/sharing/zero/hjky/rounds.go`	

Description

The Gennaro DKG and HJKY key refresh implementations do not verify the length of the Feldman verification vector, thus allowing a malicious participant to surreptitiously increase the threshold of the protocol. The Gennaro DKG uses a Pedersen commitment (whose length is also not verified) and a Feldman verification vector, both of which are verified in round 3.

```
for pid := range p.ac.Shareholders().Iter() {
	if pid == p.id {
		continue // skip myself
	}
	inB, _ := r3bi.Get(pid)
	p.state.receivedFeldmanVerificationVectors.Put(pid,
inB.FeldmanVerificationVector)

	inU, _ := r3ui.Get(pid)
	feldmanShare, _ := feldman.NewShare(inU.Share.ID(), inU.Share.Value(), nil)
	if err := p.state.feldmanVSS.Verify(feldmanShare,
inB.FeldmanVerificationVector); err != nil {
		return nil, errs.WrapIdentifiableAbort(err, pid, "failed to verify
feldman share from party %d", pid)
	}
	referencePedersenVector, _ :=
p.state.receivedPedersenVerificationVectors.Get(pid)
	if err := p.state.pedersenVSS.Verify(inU.Share, referencePedersenVector); err
!= nil {
		return nil, errs.WrapIdentifiableAbort(err, pid, "failed to verify
pedersen share from party %d", pid)
	}
	summedShareValue = summedShareValue.Add(inU.Share.Value())
	summedFeldmanVerificationVector =
summedFeldmanVerificationVector.Op(inB.FeldmanVerificationVector)
}
```

Figure 12.1: The participant verifies the received Feldman vector without checking the length. (`bron-crypto/pkg/threshold/dkg/gennaro/rounds.go#L69–L87`)

Feldman verification includes evaluation of the polynomial in the exponent.

```
func (d *Scheme[E, FE]) Verify(share *Share[FE], reference VerificationVector[E,
FE]) error {
	if reference == nil {
		return errs.NewIsNil("verification vector is nil")
	}
	x := d.shamirSSS.SharingIDToLagrangeNode(share.ID())
	yInExponent := reference.Eval(x)
	shareInExponent := d.basePoint.ScalarOp(share.Value())
	if !yInExponent.Equal(shareInExponent) {
		return errs.NewVerification("verification vector does not match share
in exponent")
	}
	return nil
}
```

Figure 12.2: Feldman verification evaluates the polynomial without verifying the length. (`bron-crypto/pkg/threshold/sharing/feldman/scheme.go#L116–L127`)

This polynomial evaluation loops through all coefficients and the length of the coefficients is not limited anywhere.

```
func (p *ModuleValuedPolynomial[ME, S]) Eval(at S) ME {
	out := p.coeffs[len(p.coeffs)-1].Clone()
	for i := len(p.coeffs) - 2; i >= 0; i-- {
		out = out.ScalarOp(at).Op(p.coeffs[i])
	}
	return out
}
```

Figure 12.3: Polynomial evaluation iterates through all coefficients. (`bron-crypto/pkg/base/polynomials/polynomial_module.go#L234–L240`)

The same verification functions are used during key refresh in the HJKY protocol.

```
if !b.VerificationVector.Coefficients()[0].Equal(p.group.OpIdentity()) ||
p.scheme.Verify(u.ZeroShare, b.VerificationVector) != nil {
	return nil, nil, errs.NewIdentifiableAbort(id, "invalid share")
}
share = share.Add(u.ZeroShare)
verificationVector = verificationVector.Op(b.VerificationVector)
p.state.verificationVectors[id] = b.VerificationVector
```

Figure 12.4: The HJKY protocol uses the same Feldman verification strategy. (`bron-crypto/pkg/threshold/sharing/zero/hjky/rounds.go#L73–L78`)

In addition to causing a DoS, this finding can enable other attacks.

Exploit Scenario
A malicious participant sends a longer Feldman verification vector during key generation or refresh. The resulting private key shares now lie on a higher-degree polynomial than intended, corresponding to a higher threshold. The honest parties cannot sign any additional messages without the help of the dishonest party.

Recommendations
Short term, verify the length of received verification vectors against the threshold from the access structure.

Long term, implement length verification for all received messages.

13. Gennaro DKG is vulnerable to rogue-key attacks by a malicious participant

Severity: **High**	Difficulty: **Medium**
Type: Cryptography	Finding ID: TOB-BRON-13
Target: `pkg/threshold/dkg/gennaro/rounds.go`	

Description

A malicious participant can perform a rogue-key attack against the Gennaro DKG algorithm by using different polynomials for the Pedersen and Feldman stages. The Gennaro DKG implementation comprises the following rounds:

1. Participants generate polynomials `f` and `f'` and perform a Pedersen commitment to the coefficients of `f` using `f'` as a witness. They multiply each coefficient of `f` by the generator `G`, and each coefficient of `f'` by the second generator `H`, and add the corresponding coefficients (i.e., of the same degree). We write the resulting vector commitment to all coefficients as `f G + f' H`.

2. Participants generate their Feldman verification polynomial by computing and broadcasting `f G`. For each participant `i`, they also send the shares `f(i)` and `f'(i)`.

3. Participants verify the Feldman vector by comparing `f(i) * G = [f G](i)` and verify the Pedersen vector by comparing `f(i) * G + f'(i) * H = [f G + f H](i)`. They then add the Feldman vectors together and use it to derive the public key.

Essentially, what happens is that the Pedersen and Feldman verifications check that evaluating the corresponding polynomials in the exponent results in the same share as that received by the participant. However, there is no proof that the polynomials `f` used for Pedersen and Feldman are the same, unless you assume an honest majority and the lengths of the Pedersen and Feldman verification vectors are strictly enforced.

In a dishonest majority setting, it is trivial for malicious participants to construct degree-`t` polynomials f_1 and f_2 that agree at all evaluation points for the honest participants (which number less than `t + 1`), but differ elsewhere. Being able to choose different polynomials allows the malicious participant to bypass the commitment requirements, which enables rogue key attacks as described below. Note that, due to finding TOB-BRON-12, this attack is also possible in an honest-majority setting, because the malicious participant can use polynomials of arbitrary degree to circumvent the fact that they need to give matching shares to all honest participants.

The Gennaro paper describing this algorithm assumes an honest majority (because robustness is impossible otherwise). However, as mentioned, in the current implementation the attack is also possible in the honest majority setting. In general, preventing rogue key attacks requires at least one of the following countermeasures:

1. Broadcasting a commitment to the constant term of the Feldman polynomial before revealing the verification vectors for all parties
2. Proving in zero-knowledge that each participant knows the constant term of their sharing polynomial

In a dishonest majority setting, or in a setting where the degrees of the Feldman or Pedersen polynomials are not limited, the Gennaro DKG fails because the Pedersen commitment does not sufficiently bind the Feldman polynomial.

Exploit Scenario

Consider a 2-out-of-2 scenario with one honest and one malicious party. Without loss of generality, the honest party has index 1. The malicious party generates a key pair `(x, X = xG)` and a share `y` for the honest party. In round 1, they generate some random polynomials `f`, `f'` such that `f(1) = y` and honestly perform the Pedersen commitment (resulting in some blinding share `f'(1) = y'`).

In round 2, they wait until the honest participant broadcasts their Feldman vector `(A, B)`. They compute `C = X - A` and `D = y*G - C`, and send `(C,D)` as their Feldman vector and `y` and `y'` as the shares.

In round 3, the honest participant verifies the Pedersen commitments, which match `y` and `y'`. Then, for Feldman verification, they calculate `y*G = C + D`, which is true by construction. They compute the final public key as `A + C = X`, which is a key controlled by the attacker.

Appendix F contains an implementation of the attack in a 3-out-of-3 setting.

Extending the scenario to an honest majority setting involves performing Lagrange interpolation in the exponent to derive the coefficients required for honest participants based on their shares, which results in a longer Feldman or Pedersen vector.

Recommendations

Short term, add zero-knowledge proofs for the constant term of the Feldman polynomial and resolve finding TOB-BRON-12.

Long term, carefully document the assumptions that each implemented protocol makes, including but not limited to assumptions regarding honest majority. Then, implement checks in the protocol creation functions that verify whether the desired access structure meets those assumptions.

14. Paillier nonces are sampled from the wrong space

Severity: **Informational**	Difficulty: **N/A**
Type: Cryptography	Finding ID: TOB-BRON-14
Target: `pkg/encryption/paillier/nonces.go`	

Description

Paillier nonces are sampled from the range $[0, n^2)$ rather than $[0, n)$. Additionally, there is no check that the sampled nonce r is coprime with n.

Paillier encryption is based on the homomorphism from $(\mathbb{Z}/n\mathbb{Z}) \times (\mathbb{Z}/n\mathbb{Z})^*$ to $(\mathbb{Z}/n^2\mathbb{Z})$, given by $(m, r) \mapsto (1 + N)^m r^n$. To encrypt a value m in $(\mathbb{Z}/n\mathbb{Z})$, you select a uniformly random element r in $(\mathbb{Z}/n\mathbb{Z})^*$ and then compute the ciphertext as $c = (1 + N)^m r^N$. In the implementation of Paillier encryption, nonces are sampled by calling the `NonceSpace.Sample` method.

```
func (ns *NonceSpace) Sample(prng io.Reader) (*Nonce, error) {
	u, err := ns.g.Random(prng)
	if err != nil {
		return nil, errs.WrapRandomSample(
			err, "failed to sample from nonce space")
	}
	return &Nonce{u: u}, nil
}
```

Figure 14.1: `NonceSpace.Sample` calls `ns.g.Random` to generate a new nonce.
(`bron-crypto/pkg/encryption/paillier/nonces.go#L28–L34`)

Here, the call to `ns.g.Random(prng)` returns a uniformly random element from the group `ns.g`. However, `ns.g` is given by `znstar.NewPaillierGroupOfUnknownOrder(n2, n)`, which is equal to $(\mathbb{Z}/n^2\mathbb{Z})$ rather than $(\mathbb{Z}/n\mathbb{Z})$.

```
func NewPaillierGroupOfUnknownOrder(n2, n *num.NatPlus)
	(*PaillierGroupUnknownOrder, error)
{
	if n2.AnnouncedLen() < 4096 {
		return nil, errs.NewValue("modulus must be at least 4096 bits")
	}
	if !n.Mul(n).Equal(n2) {
		return nil, errs.NewValue("n isn't sqrt of n")
	}
	zMod, err := num.NewZMod(n2)
	if err != nil {
			return nil, errs.WrapFailed(err, "failed to create ZMod")
		}
		arith, ok := modular.NewSimple(zMod.Modulus().ModulusCT())
		if ok == ct.False {
			return nil, errs.NewFailed("failed to create SimpleModulus")
		}

		return &PaillierGroupUnknownOrder{
			DenseUnitGroupTrait: DenseUnitGroupTrait[
				*modular.SimpleModulus,
				*PaillierGroupElement[*modular.SimpleModulus],
				PaillierGroupElement[*modular.SimpleModulus]
			]{
				zMod:  zMod,
				arith: arith,
				n:     n,
			},
		}, nil
}
```

Figure 14.2: The Paillier group is created over the $\mathbb{Z}$-module $(\mathbb{Z}/n^2\mathbb{Z})$.
(bron-crypto/pkg/base/nt/znstar/paillier.go#L57–L80)

During encryption, the nonce r is implicitly reduced modulo n, since if $r = an + b$ with $b < n$, we have $(1+N)^m r^n = (1+N)^m (an+b)^n = (1+N)^m (b^n + \ldots) = (1+N)^m b^n$ by the binomial theorem. Since the pre-image of the map $x \mapsto x \bmod n$ from $[0, n^2)$ to $[0, n)$ always has size n, sampling r uniformly from $[0, n^2)$ always results in a uniform element b in $[0, n)$.

There is also no check that $\gcd(r, n) = 1$ to ensure that the sampled nonce r is in the multiplicative group $(\mathbb{Z}/n\mathbb{Z})^*$. If n is chosen honestly, the probability that $\gcd(r, n) > 1$ is vanishingly small. However, in protocols like Lindell17, you encrypt values to the other participant and we cannot assume that n is honestly generated. In fact, the Lindell17 paper specifically states that Bob must check whether $\gcd(r, n) = 1$ whenever encrypting values. We have not found a way to exploit this to leak information about key shares or forge signatures.

Recommendations

Short term, sample nonces from the correct range and ensure that they are coprime with the modulus n.

Long term, add tests to ensure that random values are sampled from the correct sets.

15. The L_{PDL} verifier accepts forged proofs

Severity: **Low**	Difficulty: **Medium**
Type: Cryptography	Finding ID: TOB-BRON-15
Target: `pkg/proofs/paillier/lpdl/rounds.go`	

Description

The implementation of the L_{PDL} verifier samples the value b from $\mathbb{Z}/q\mathbb{Z}$ rather than $\mathbb{Z}/q^2\mathbb{Z}$ in round one of the protocol. This allows a malicious prover to forge L_{PDL} proofs.

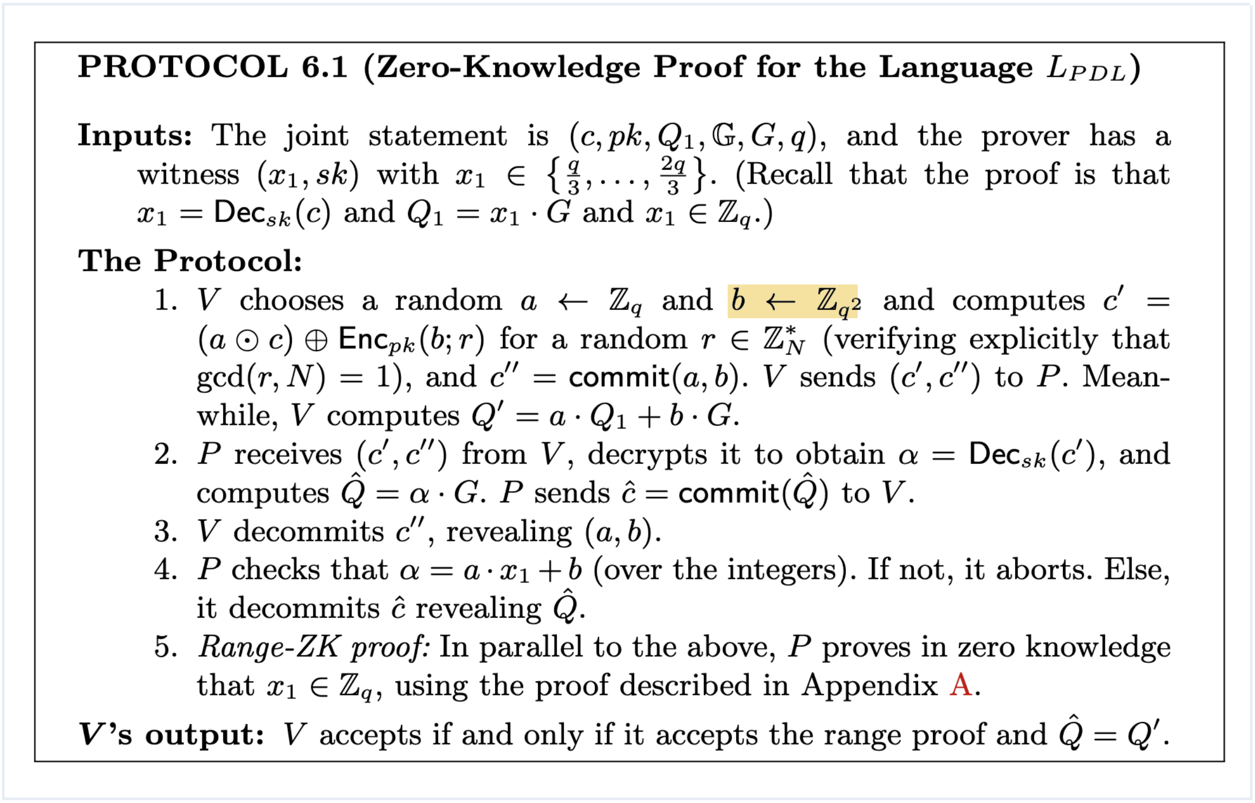
```
// 1. choose random a, b (both from Z/qZ since they're used as curve scalars)
verifier.state.a, err = verifier.state.zModQ.Random(verifier.prng)
if err != nil {
	return nil, errs.WrapFailed(err, "cannot generate random integer")
}
verifier.state.b, err = verifier.state.zModQ.Random(verifier.prng)
if err != nil {
	return nil, errs.WrapFailed(err, "cannot generate random integer")
}
```

Figure 15.1: The random value b is sampled from the wrong set in the `Verifier.Round1` method. (`bron-crypto/pkg/proofs/paillier/lpdl/rounds.go#L16–L24`)

To see how this can be used to forge L_{PDL} proofs, let x be the prover's key share. Suppose that a malicious prover sends $\mathrm{Enc}(y)$ for some $y < q$ to the verifier and wants to convince her that this is a correct encryption of x.

In round 1, the L_{PDL} verifier samples random values $a \leftarrow \mathbb{Z}/q\mathbb{Z}$ and $b \leftarrow \mathbb{Z}/q\mathbb{Z}$, computes $c' = \mathrm{Enc}(ay{+}b)$ from $\mathrm{Enc}(y)$, and then sends c' to the prover. In round 2, the prover decrypts c' to get $\alpha = ay + b$. (Note that α has not been reduced modulo the Paillier modulus m since $2q + q << m$.)

Now, if q is an n-bit integer, then y must be an n-bit integer due to the range proof. Therefore, ay has $2n$ bits, but b has only n bits. Since y is known to the prover, this means that the high n bits of $ay + b$ provide n linear equations for the n bits in a plus one carry bit from b. The prover can solve these to recover a and then compute b as $b = \alpha - ay$. The recovered values a and b will be correct with probability $1/2$ (if the prover guesses the carry bit correctly). Having recovered a and b, the prover can compute $Q\hat{} = (ax{+}b)G$ using their private key share x. Since $Q\hat{} = Q'$, the verifier accepts as long as the corresponding range proof verifies. Note that, because the range proof is also forgeable (see TOB-BRON-23), Alice can encrypt a larger y, which would allow her to forge the proof without having to guess the carry bit from b.



PROTOCOL 6.1 (Zero-Knowledge Proof for the Language L_{PDL})

Inputs: The joint statement is $(c, pk, Q_1, \mathbb{G}, G, q)$, and the prover has a witness (x_1, sk) with $x_1 \in \left\{\frac{q}{3}, \ldots, \frac{2q}{3}\right\}$. (Recall that the proof is that $x_1 = \mathsf{Dec}_{sk}(c)$ and $Q_1 = x_1 \cdot G$ and $x_1 \in \mathbb{Z}_q$.)

The Protocol:

1. V chooses a random $a \leftarrow \mathbb{Z}_q$ and $b \leftarrow \mathbb{Z}_{q^2}$ and computes $c' = (a \odot c) \oplus \mathsf{Enc}_{pk}(b; r)$ for a random $r \in \mathbb{Z}_N^*$ (verifying explicitly that $\gcd(r, N) = 1$), and $c'' = \mathsf{commit}(a, b)$. V sends (c', c'') to P. Meanwhile, V computes $Q' = a \cdot Q_1 + b \cdot G$.
2. P receives (c', c'') from V, decrypts it to obtain $\alpha = \mathsf{Dec}_{sk}(c')$, and computes $\hat{Q} = \alpha \cdot G$. P sends $\hat{c} = \mathsf{commit}(\hat{Q})$ to V.
3. V decommits c'', revealing (a, b).
4. P checks that $\alpha = a \cdot x_1 + b$ (over the integers). If not, it aborts. Else, it decommits $\hat{c}$ revealing $\hat{Q}$.
5. *Range-ZK proof:* In parallel to the above, P proves in zero knowledge that $x_1 \in \mathbb{Z}_q$, using the proof described in Appendix A.

***V*'s output:** V accepts if and only if it accepts the range proof and $\hat{Q} = Q'$.

Figure 15.2: The value b is sampled from $\mathbb{Z}/q^2\mathbb{Z}$ in the zero-knowledge proof for L_{PDL} to mask the product ax_1 which has size $\sim q^2$.

Exploit Scenario

A malicious prover uses the method described above to forge the L_{PDL} proof in Lindell17. The verifier is convinced that they have a correct encryption of the prover's private key share. When the prover and verifier attempt to sign messages using the Lindell17 signing protocol, this fails. Since the forged L_{PDL} proof verifies as correct, the verifier is mistakenly identified as malicious.

Recommendations

Short term, sample b from $\mathbb{Z}/q^2\mathbb{Z}$ rather than $\mathbb{Z}/q\mathbb{Z}$ in round one of the L_{PDL} proof verifier.

Long term, add tests to ensure that all random values are sampled from the correct sets.

16. The library generates zero-sharing IDs

Severity: **Informational**	Difficulty: **N/A**
Type: Cryptography	Finding ID: TOB-BRON-16
Target: `pkg/threshold/sharing/utils.go`	

Description

The library defaults to generating zero-sharing IDs when performing Shamir Secret Sharing, which means that the participant with ID 0 learns the full secret, not just their share. This issue existed in commit `e01154` when testing began, but it is patched in commit `4dfa2c` along with the fix to TOB-BRON-11.

The `NewOrdinalShareholderSet` utility function in the threshold module generates a set of shareholder IDs, but any positive `count` results in an `ID` with value 0.

```
func NewOrdinalShareholderSet(count uint) ds.Set[ID] {
	out := hashset.NewComparable[ID]()
	for i := range count {
		out.Add(ID(i))
	}
	return out.Freeze()
}
```

Figure 16.1: The generated shareholder set includes ID 0 in commit `e01154`.
(`bron-crypto/pkg/threshold/sharing/utils.go#L11–L17`)

In the Shamir Secret Sharing scheme, the dealer constructs a polynomial f where the shared secret is $f(0)$ and the participant with ID i receives the share $f(i)$. If $i = 0$, then that participant learns the shared secret, violating threshold guarantees.

Recommendations

Short term, follow the recommendations in TOB-BRON-11 and increment `i` before creating an `ID`. Commit `4dfa2c` adds 1 to `i`, satisfying this recommendation.

Long term, add tests to ensure that this function does not return participants with ID 0.

17. BLS key generation does not match standard recommendations

Severity: **Informational**	Difficulty: **N/A**
Type: Cryptography	Finding ID: TOB-BRON-17
Target: `pkg/signatures/bls/core.go`	

Description

The IRTF standard draft for BLS signatures includes a recommended method for generating a BLS key from a secret byte string. The library implements this functionality in `generateWithSeed`, but the implementation differs slightly from the standard draft. Even though variations from the recommendation are standard-compliant and do not appear to introduce vulnerabilities, it appears that these deviations are unintentional.

The `generateWithSeed` function in the BLS module has similar functionality to the `KeyGen` algorithm in the standard draft.

```
// step 2.3.1
for d.IsZero() {
        ikm = append(ikm, 0)
        // step 2.3.2
        kdf := hkdf.New(
                hashing.HashFuncTypeErase(RandomOracleHashFunction),
                ikm,
                salt,
                []byte{0, 48}) // TODO: make sure this is correct
        okm := make([]byte, sf.WideElementSize())
        // Leaves key_info parameter as the default empty string
        // step 2.3.3
        if _, err := io.ReadFull(kdf, okm[:48]); err != nil {
                return *new(S), *new(K), errs.WrapRandomSample(
                        err, "could not read from KDF")
        }
        copy(okm[:48], sliceutils.Reversed(okm[:48]))

        // step 2.3.4
        d, err = sf.FromWideBytes(okm)
```

Figure 17.1: Steps 1 and 2 in `generateWithSeed`
(`bron-crypto/pkg/signatures/bls/core.go#L37–L51`)

```
1. while True:
2.     PRK = HKDF-Extract(salt, IKM || I2OSP(0, 1))
3.     OKM = HKDF-Expand(PRK, key_info || I2OSP(L, 2), L)
4.     SK = OS2IP(OKM) mod r
```

Figure 17.2: Corresponding steps in KeyGen (BLS IRTF draft)

Note that in the former, an additional `0` byte is appended for each iteration, and in the latter, exactly one `0` byte is appended on every iteration. Also note that in the former, the first 48 of 64 bytes are reversed and `FromWideBytes` calculates `d` expecting big-endian order. In the latter, `OS2IP` calculates `SK` expecting big-endian order. These two calculations differ, but the resulting private keys have the same entropy and are equally secure.

Recommendations

Short term, fix the implementation to align with the standard draft, or document the deviations.

Long term, add tests to check that key generation agrees with the recommended draft standard.

18. The pailliern proof does not incorporate statement

Severity: **Informational**	Difficulty: **N/A**
Type: Cryptography	Finding ID: TOB-BRON-18
Target: `pkg/proofs/paillier/pailliern/pailliern.go`	

Description

The `NewProver` function creates a new `Prover` instance, initializing the Fiat-Shamir transcript by writing a set of fixed strings, followed by the session ID. That transcript is then immediately used to generate challenge values using the `extractRhos` method. The Paillier modulus `N` is not incorporated into the transcript.

This means that an attacker with foreknowledge of the session ID would know the output of the transcript challenge generation process when selecting their modulus `N`.

One can argue that the output of `extractRhos` *loosely* depends on `N` because it uses a rejection-sampling technique. However, in most cases, this means that `extractRhos` will return the same challenge values for `N` and `N + 1`, since the chance of a challenge value `c_i` being rejected for `N + 1` and not `N` is minimal. This, at the very least, breaks the definition of a "random oracle" as it is used in `pkg/proofs/paillier/pailliern/README.md`.

We consider this finding informational severity because the code in the `pailliern` module is currently unused.

Recommendations

Short term, incorporate the modulus into the transcript.

Long term, if the `pailliern` package is no longer needed, consider removing it from the repository.

19. Paillier range proof serialization problems	
Severity: **Low**	Difficulty: **Medium**
Type: Cryptography	Finding ID: TOB-BRON-19
Target: `pkg/proofs/paillier/range/range.go`	

Description

The range proof implementation has several different methods for serializing commitments, statements, and responses. The `SerializeStatement`, `SerializeCommitment`, and `SerializeResponse` methods (which appear unused) serialize integers into byte streams that are joined together with fixed separator bytes, while the `Statement.Bytes`, `Commitment.Bytes`, and `Response.Bytes` methods dispense with the separators.

Neither approach is suitable in a noninteractive zero-knowledge context, as they can both lead to ambiguous transcripts. As an example, if a `Commitment` struct's `C1` member were the sequence of integers `{0x1000_0000, 0x1000, 0x50}`, the serialization would be the same as if the `C1` member were `{0x10_000, 0x00, 0x100050}`. An attacker looking to forge proofs could generate a sequence of inputs, compute the associated challenge value, then move the boundaries between the elements of `C1` and `C2` to find values most favorable to forgery.

Values added to a Fiat-Shamir or Fischlin transcript should be processed in a way that makes collisions computationally infeasible to find. The most straightforward way to do this is to ensure that the serializations can be unambiguously deserialized, which can be accomplished by adopting a tag-length-value encoding scheme. Alternatively, the hash of such a serialization will work just fine, making the use of `hagrid` transcripts an attractive option.

Exploit Scenario

An attacker who wishes to cause an abort could generate a valid Paillier range proof, take advantage of the serialization malleability to create an invalid proof that serializes to the same values, and send the invalid proof to another party. The other party will reject the proof, causing an abort. The attacker can then present the valid proof to an observer as evidence that it was an honest participant and that the other party rejected the proof without reason.

Recommendations

Short term, implement better serialization for the statements, commitments, and responses.

20. Public key material equality test does not ensure equality

Severity: **Informational**	Difficulty: **N/A**
Type: Data Validation	Finding ID: TOB-BRON-20
Target: `pkg/threshold/tsig/tschnorr/tschnorr.go`	

Description

The equality test for the `BasePublicMaterial` only checks that the partial public keys of the left-hand side (`spm`) form a subset of the partial public keys on the right-hand side (`other`), but fails to ensure that the two sets are equal.

```
func (spm *BasePublicMaterial[E, S]) Equal(other *BasePublicMaterial[E, S]) bool {
	if spm == nil || other == nil {
		return spm == other
	}
	for id, pk := range spm.partialPublicKeys.Iter() {
		otherPk, exists := other.partialPublicKeys.Get(id)
		if !exists || !pk.Equal(otherPk) {
			return false
		}
	}
	return spm.accessStructure.Equal(other.accessStructure) &&
		spm.fv.Equal(other.fv)
}
```

Figure 20.1: The loop ensures that all partial public keys in spm are also in `other`, but fails to ensure the converse. (`bron-crypto/pkg/threshold/tsig/base_shard.go#L53–L65`)

To identify variants of this issue, we wrote a Semgrep rule identifying implementations of `Equal` where one field is checked for set inclusion but not equality. This identified two additional implementations of `Equal` with the same issue.

- `PublicMaterial.Equal` (`pkg/threshold/tsig/tbls/tbls.go`)
- `AuxiliaryInfo.Equal` (`pkg/threshold/tsig/tecdsa/lindell17/shard.go`)

Recommendations

Short term, add a check ensuring that the sizes of the two partial public key maps are equal to `BasePublicKeyMaterial.Equal`.

Long term, review implementations of `Equal` to ensure that all relevant fields are checked for equality.

21. Inconsistent Paillier plaintext representations

Severity: **Informational**	Difficulty: **N/A**
Type: Data Validation	Finding ID: TOB-BRON-21
Target: `pkg/encryption/paillier/plaintexts.go`	

Description

Paillier plaintexts are represented as integers in the symmetric interval $[-(n - 1) / 2, (n - 1) / 2)$ by default. However, `Plaintext.Sample` can also produce plaintexts where the underlying value is sampled from a given range. Such plaintexts are not normalized to the symmetric interval $[-(n - 1) / 2, (n - 1) / 2)$.

```
func (pts *PlaintextSpace) Sample(
	lowInclusive, highExclusive *Plaintext, prng io.Reader) (*Plaintext, error)
{
	if lowInclusive == nil && highExclusive == nil {
		sampled, err := (*num.ZMod)(pts).Random(prng)
		if err != nil {
			return nil, errs.WrapRandomSample(
				err, "failed to sample from plaintext space")
		}
		v, err := num.Z().FromUintSymmetric(sampled)
		if err != nil {
			return nil, errs.WrapFailed(
				err, "failed to create centred plaintext from nat")
		}
		return &Plaintext{
			v: v,
			n: pts.N(),
		}, nil
	}
	if lowInclusive != nil && highExclusive != nil {
		// [...]
	}
	return nil, errs.NewFailed("must either be closed or open interval sampling")
}
```

Figure 21.1: Sampling a Paillier plaintext returns a value in the symmetric interval $[-(n - 1) / 2, (n - 1) / 2)$. (`bron-crypto/pkg/encryption/paillier/plaintexts.go#L34–L60`)

Additionally, the `Plaintext.isValid` method only checks that the absolute value of the plaintext is smaller than n, rather than $(n - 1) / 2$. The `Plaintext.isValid` method calls `x.v.InRange(cp.n)` to ensure that the plaintext x lies in the correct range.

```
func (cp *Plaintext) isValid(x *Plaintext) {
	if x == nil {
		panic("cannot operate on nil centred plaintexts")
	}
	if !cp.n.Equal(x.n) {
		panic("cannot operate on centred plaintexts with different moduli")
	}
	if !x.v.IsInRange(cp.n) {
		panic("cannot operate on centred plaintexts with values out of range")
	}
}
```

Figure 21.2: `Plaintext.isValid` calls `Int.InRange` to ensure that the input lies in the correct range. (`bron-crypto/pkg/encryption/paillier/plaintexts.go#L124–L134`)

However, `Int.InRange` only ensures that the absolute value of the input x is less than the modulus `cp.n`.

```
func (i *Int) IsInRange(modulus *NatPlus) bool {
	if modulus == nil {
		panic("argument is nil")
	}
	return modulus.ModulusCT().IsInRange(i.Abs().v) == ct.True
}
```

Figure 21.3: Calling `Int.Inrange` amounts to checking the absolute value of x. (`bron-crypto/pkg/base/nt/num/z.go#L372–L377`)

This means that plaintexts outside the interval $[-(n - 1)/2, (n - 1)/2)$ will be accepted as valid by `Plaintext.isValid`.

The `Plaintext.isValid` method is only used in the implementations of `Plaintext.Add` and `Plaintext.Sub`, where both inputs are normalized to the correct ranges before they are used.

In total, this means that there are essentially three different notions of what it means to be a valid representation of a Paillier plaintext.

1. An integer in the symmetric interval $[-(n - 1) / 2, (n - 1) / 2)$ (default version of `Plaintext.Sample` and the output from `Plaintext.Add` and `Plaintext.Sub`)
2. Any integer (returned by `Plaintext.Sample` with explicit bounds)
3. An integer in the symmetric interval $(-n, n)$ (integers for which `Plaintext.isValid` returns true)

Recommendations

Short term, unify the underlying representations of Paillier plaintexts, and consider removing the call to `Plaintext.isValid` from `Plaintext.Add` and `Plaintext.Sub`. This will not affect the output since the inputs are already normalized before they are used.

Long term, ensure that the underlying representation of each primitive type is well defined and specified.

22. Ineffective input validation in L_{PDL} verifier

Severity: **Informational**	Difficulty: **N/A**
Type: Data Validation	Finding ID: TOB-BRON-22
Target: `pkg/base/nt/crt/crt_multi.go`	

Description

When a new L_{PDL} verifier is created, the implementation calls `validateVerifierInputs` to validate the public inputs to the verifier. The encrypted key share $Enc(x)$ is checked to ensure that it is neither 0 or 1, and if it is, an error is returned. However, because of an implementation mistake, the code actually checks if $Enc(x)$ is both zero and one at the same time, which will never happen.

```
if v := xEncrypted.ValueCT(); v.IsZero()&v.Equal(numct.NatOne()) == ct.True {
      return errs.NewArgument("xEncrypted is invalid: %s", v.String())
}
```

Figure 22.1: The verifier checks if $Enc(x)$ is equal to zero and equal to one at the same time.
(bron-crypto/pkg/proofs/paillier/lpdl/participants.go#L178–L180)

Since one is in $(\mathbb{Z}/n^2\mathbb{Z})^*$, it is a valid ciphertext and should not be rejected by the implementation. If $Enc(x)$ is zero, then $Enc(ax + b)$ will also be zero, for all values a and b. This means that a malicious prover who sends $Enc(x) = 0$ to the verifier does not learn a and b, and will not be able to complete the proof.

Recommendations

Short term, remove the check ensuring that the ciphertext is not one from the `validateVerifierInputs` function.

Long term, rely on the underlying types to verify correctness properties of their corresponding values. For example, since $Enc(x)$ is a `paillier.Ciphertext`, the fact that it is nonzero should be checked when the value is created, since zero is not in $(\mathbb{Z}/n^2\mathbb{Z})^*$.

23. Malicious Alice can use invalid Paillier ciphertext to recover Lindell17 key	
Severity: **High**	Difficulty: **Medium**
Type: Cryptography	Finding ID: TOB-BRON-23
Target: `pkg/threshold/tsig/tecdsa/lindell17/keygen/dkg/round.go`	

Description

Bob does not verify that received nonces, received ciphertexts, or generated ciphertexts are coprime with Alice's Paillier modulus during the Lindell17 DKG protocol, which allows a malicious Alice to recover the private key.

In the implementation, Alice splits her Shamir share x_1 into two components `x'` and `x''`, encrypts both using a freshly generated Paillier key pair, and proves to Bob that these are correct encryptions of the scalars of her corresponding elliptic curve public keys using the L_{PDL} proof, which includes a range proof. This range proof verifies among other things that Alice provided a well-formed Paillier ciphertext.

However, Bob does not check any of the following:

1. Nonces received from Alice are nonzero.
2. Ciphertexts received from Alice are nonzero.
3. Ciphertexts created by Bob are nonzero.
4. Nonces received from Alice are coprime to N.
5. Ciphertexts received from Alice are coprime to N.
6. Ciphertexts created by Bob are coprime to N.

Note that checks 4, 5, and 6 encompass checks 1, 2, and 3, respectively. Because Bob does not check 1 and 2, Alice can arbitrarily forge the range proof with any input, enabling Alice to obtain the full private key share using a well-formed Paillier key. Even if Bob were to check 1 and 2, the fact that Bob does not check 5 and either 3 or 4 allows Alice to forge the range proof with a malformed Paillier key.

Essentially, the attack works by creating a malformed Paillier encryption, bypassing the range proof by creating ciphertexts that are zero or that will become zero when combined, and using the malformed encryption to obtain any exponent that Bob adds to the ciphertext via homomorphic multiplication by a constant. Since the malformed encryption still allows Alice to decrypt normally, she can perform the rest of the L$_{PDL}$ proof honestly, meaning that she does not need to rely on the fact that it is forgeable (TOB-BRON-15). Note that Alice can choose arbitrarily large Paillier keys, as Bob only verifies that the length meets a lower bound.

Exploit Scenario

Assuming that Bob does not check that received nonces and ciphertexts are nonzero, a malicious Alice takes the following steps during the DKG:

1. Generate a prime p_0, and a special modulus $N = p_1 p_2$ such that $gcd(N, \varphi(N)) = 1$, where $p_2 > q^3$ is a large prime such that $(p_2, 1 + N)$ is a valid Paillier public key, and $p_0, p_1 > q$ are two large primes such that (p_1, p_0) is a valid Paillier public key, where q is the curve order. For example, using one 1024-bit prime p_2 and two 1024-bit primes p_0, p_1 is sufficient for a 256-bit curve, but Alice can choose larger keys if desired.

2. Calculate $c_{key} = g^x \cdot p_0^{p_2} \bmod N^2$ where $g = 1 + N$ and send it to Bob.

3. In the range proof, send $c_j^i = 0$ for all i, j, and respond with $r_j^i = 0$ regardless of challenge, where the corresponding other values can be any legal values that will pass the checks by Bob, as encrypting using nonce zero will always match the committed zero ciphertext.

4. During L$_{PDL}$, receive $c' = c^a \cdot g^b \cdot r^N \bmod N^2 = g^{ax+b} \cdot r^N p_0^{a \cdot p_2}$ from Bob, who has generated $a < q$, $b < q^2$.

5. Calculate $c' \bmod p_2^2 = g^{ax+b} \cdot (r^{N/p_2} p_0^a)^{p_2} \bmod p_2^2$ and decrypt it as a Paillier-$(p_2, 1 + N)$ ciphertext to obtain $(ax + b) \bmod p_2$ which is equal to $ax + b$ over the integers. At this point, Alice can complete the rest of the L$_{PDL}$ proof, but in fact, she can recover a as follows.

6. Calculate $(c' \cdot g^{-(ax+b)} \bmod N^2) \bmod p_1^2 = p_0^{a \cdot p_2} \cdot (r^{N/p_1})^{p_1} \bmod p_1^2$ and decrypt it as a Paillier-(p_1, p_0) ciphertext to obtain $a \cdot p_2 \bmod p_1$.

7. Multiply the result by $p_2^{-1} \bmod p_1$ to recover $a \bmod p_1$ which is equal to a over the integers.

8. Alice can now recover b from $ax + b$ and use a and b to complete the rest of the L$_{PDL}$ protocol honestly (or just use the $ax + b$ value from step 4 directly).

Note that the attack requires Bob to generate his Paillier randomness r coprime to N in step 3 and the probability of this depends on the primes that Alice chooses (due to finding TOB-BRON-15). As Alice uses large primes, this probability is negligible.

In the subsequent signing protocol, Bob computes $g^{k_2^{-1} \cdot (m' + x\,r) + \rho\, q} \cdot p_0^{p_2 \cdot k_2^{-1} \cdot \lambda_1 \cdot r} \cdot r_1^N \cdot r_2^N$, where λ_1 is the Lagrange interpolation value for Alice's share, x is the combined private key, and r_1 and r_2 are random values generated by Bob (and hence coprime with the modulus with overwhelming probability). Now, Alice can repeat steps 4–6 to recover $k_2^{-1} \cdot \lambda_1 \cdot r \; mod\; q$ and hence k_2, as well as the decrypted value corresponding to the partial signature. Alice can recover the private key from the partial signature and the partial nonce, or complete the signature and recover the private key from the complete nonce.

In the case that Bob checks that received nonces and ciphertexts are nonzero but does not perform the other checks that received and generated values are coprime to Alice's Paillier modulus, Alice can modify the attack as follows:

1. Alice generates her Paillier key as $N = p_0 p_1 p_2$ using the same constraints on the primes as above, and calculates c_{key} as before (which means it is no longer coprime with N).
2. In the range proof, Alice generates valid w_j^i-values and encrypts them using $r_j^i = (p_1 p_2)^y$ for some small random y.
3. When the challenge bit is zero, Alice provides these nonzero nonces that are not coprime to her modulus to Bob who is successfully able to re-encrypt.
4. When the challenge bit is one, Alice provides nonces r_i that are multiples of N, which will lead to a ciphertext of zero when Bob re-encrypts the provided values. However, $c_{key} \cdot c_i = g^x \cdot p_0^{p_2} \cdot g^{w_j^i} (p_1 p_2)^{y \cdot N} = g^x \cdot p_0^{p_2 - 2} \cdot g^{w_j^i} (p_1 p_2)^{y \cdot N - 2} \cdot N^2 = 0 \; mod\; N^2$, which will match the ciphertext produced by Bob.

The rest of the attack works as before.

Recommendations

Short term, verify that Paillier ciphertexts and nonces are coprime with the modulus when parsing them.

Long term, add negative tests for all explicitly specified checks in the protocols.

24. Paillier range proof verifier fails to check input lengths	
Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-BRON-24
Target: `pkg/proofs/paillier/range/range.go`	

Description
The `Verify` method does not check that the `response.W1`, `response.W2`, `response.R1`, `response.R2`, `response.Wj`, `response.J`, `commitment.C1`, or `commitment.C2` arrays have the correct length. If any of these arrays are shorter than expected, the `Verify` method will panic with an out-of-range array access.

Exploit Scenario
An attacker who wishes to cause a DoS participates in the protocol and then sends a malformed commitment or response in which one of the arrays is shorter than expected.

Recommendations
Short term, add checks to `Verify` to ensure that all arrays accessed have the correct length, returning an error if one or more of the checks fails.

Long term, examine all handling of untrusted and adversarial data to ensure that data is being validated for form and basic correctness before processing.

25. Insufficient test coverage

Severity: **Low**	Difficulty: **Undetermined**
Type: Testing	Finding ID: TOB-BRON-25
Target: `pkg/*`	

Description

Most components in the codebase have corresponding unit tests, and many also implement some form of negative testing. However, by reviewing the reports generated by the Go `cover` tool, we found that negative testing could be improved across most components. We also found components that completely lack testing. In particular, there are no tests that exercise the Fiat-Shamir compiler (`pkg/proofs/sigma/compiler/fiatshamir`) or the polynomial implementation used for secret sharing and polynomial interpolation (`pkg/base/polynomials`).

Exploit Scenario

A vulnerability is introduced by mistake when the corresponding component is refactored. Because the component is not tested sufficiently, the vulnerable code is deployed in a production environment.

Recommendations

Short term, review test coverage and aim to have coverage of all nontrivial functions.

Long term, use randomized testing (e.g., property testing with rapid) to improve negative test coverage, and consider using tools like `go-test-coverage` to enforce minimum test coverage in CI/CD.

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. Code Quality Issues

This appendix contains findings that do not have immediate or obvious security implications. However, addressing them may enhance the code's readability and may prevent the introduction of vulnerabilities in the future.

- **The L_{PDL} prover disallows $Q\hat{} = 0$ for no reason.** The L_{PDL} verifier aborts in round four if $Q\hat{}=0$. However, there is nothing special about the identity here, and there is no way for the prover to forge proofs with $Q\hat{}=0$ since the verifier also checks that $Q\hat{}=aQ_1 + bG$. (In fact, the verifier can force $Q\hat{}=0$ by choosing $a = b = 0$ in round one.)

- **Pedersen witness construction does not use the corresponding function.** When re-randomizing Pedersen commitments, a random witness is constructed directly from a value, instead of using the function `NewWitness`.

```
witness := &Witness[S]{v: wv}
```

Figure B.1: `bron-crypto/pkg/commitments/pedersen/commitment.go#L80`

- **The implementation of the Paillier `SelfEncrypter` could reuse the implementation of the Paillier `Encrypter`.** This would avoid code duplication and make the implementation easier to maintain.

- **The Fischlin verifier reimplements the hash function used for the proof of work.** It could reuse the implementation in `utils.go` to avoid code duplication.

- **The BIP0340 batch verifier checks that each public key is different from the identity twice.**

```
if sliceutils.Any(publicKeys, func(pk *PublicKey) bool {
        return pk == nil ||
        pk.Value() == nil ||
        pk.Value().IsOpIdentity() ||
        pk.Value().IsOpIdentity()
}) {
        return errs.NewArgument("some public keys are nil or identity")
}
```

Figure B.2: `pkg/signatures/schnorrlike/bip340/bip340.go#L313-L315`

- **Ad-hoc construction of constant to become a valid multiple.** The `xi` parameter is constructed from kappa and the collision resistance constant. By coincidence, this results in a correct multiple of the value that SoftSpokenOT requires. However, `xi` should be constructed by adding the statistical security and then rounded up to the correct multiple instead.

```
xi := kappa + base.CollisionResistance // normally this should be statistical
    security, but then xi is an invalid parameter for softspoken
```

Figure B.3: *bron-crypto/pkg/threshold/rvole/softspoken/participant.go#L61*

- **Incorrect panic message.** The error message says hash to curve, but the operation is hash to field.

```
for l, message := range messages[0] {
        alice.alpha[xi][0][l], err = alice.suite.field.Hash(message)
        if err != nil {
                return nil, nil, errs.WrapFailed(err, "cannot hash to curve
message")
        }
}
for l, message := range messages[1] {
        alice.alpha[xi][1][l], err = alice.suite.field.Hash(message)
        if err != nil {
                return nil, nil, errs.WrapFailed(err, "cannot hash to curve
message")
        }
    }
```

Figure B.4: *bron-crypto/pkg/threshold/rvole/softspoken/rounds.go#L65–L76*

- **Partial signatures store shards of R when the final value is available.** The partial signature that each participant stores contains their shard of the nonce commitment R. However, at this point in the protocol, all participants have a consensus on the value of the nonce commitment and each participant has proven that their shard is correct. There is no reason to recombine the shards later during aggregation as opposed to storing the final nonce commitment.

```
partialSignature, err =
    dkls23.NewPartialSignature(c.state.bigR[c.shard.Share().ID()], u, w)
```

Figure B.5: *bron-crypto/pkg/threshold/tsig/tecdsa/dkls23/signing/interactive/sign/rounds.go#L281*

- **Fixed values and strings should be defined as const.** Various fixed values should be defined as `const` values for better maintainability.

```
h, err := ts.Extract(tape, "second generator of pedersen key", group)
```

Figure B.6: *bron-crypto/pkg/threshold/dkg/gennaro/participant.go#L108*

```
c.tape.AppendBytes("prover", proverIDBytes)
```

Figure B.7: bron-crypto/pkg/threshold/tsig/tecdsa/lindell17/signing/rounds.go#L237

```
tape.AppendBytes("prover", proverIDBytes)
tape.AppendBytes("bigQTwin", bigQTwin.ToCompressed())
```

Figure B.8: bron-crypto/pkg/threshold/tsig/tecdsa/lindell17/keygen/dkg/round.go#L446–L447

- **Adding shares does not check that IDs match.**

```
func (s *Share[FE]) Add(other *Share[FE]) *Share[FE] {
	return &Share[FE]{
		id: s.id,
		v:  s.v.Add(other.v),
	}
   }
```

Figure B.9: bron-crypto/pkg/threshold/sharing/shamir/share.go#L74–L79

- **Lindell17 signature Paillier calculations can be made more efficient.** Currently, Bob calculates the partial signature from scratch. If Bob precomputes and stores cKey$^{\lambda 1}$·Enc(additiveShare), it will save four field multiplications, one Paillier encryption, and one Paillier ciphertext multiplication for every signature. The precomputed value is fixed per primary participant.

```
// c2 = Enc(k2^(-1) * r * (cKey * λ1 + share * λ2))
c2LeftExponent, err := num.N().FromBytes(k2Inv.Mul(r).Mul(lambda1).Bytes())
if err != nil {
	return nil, errs.WrapFailed(err, "cannot convert c2 left exponent to
Nat")
}
c2Left := cKey.ScalarExp(c2LeftExponent)

c2RightMessage, err :=
pk.PlaintextSpace().FromBytes(k2Inv.Mul(r).Mul(additiveShare).Bytes())
if err != nil {
	return nil, errs.WrapFailed(err, "cannot convert c2 right message to
plaintext")
}
c2Right, _, err := enc.Encrypt(c2RightMessage, pk, prng)
if err != nil {
	return nil, errs.WrapFailed(err, "cannot encrypt c2")
}
    c2 := c2Left.Mul(c2Right)
```

Figure B.10: *`bron-crypto/pkg/threshold/tsig/tecdsa/lindell17/signing/signing.go#L62–L77`*

- **The Lindell17 trusted dealer generates too long Paillier keys.** When generating Paillier keys, the Lindell17 trusted dealer generates both primes using the length of the final key, resulting in a Paillier key that is twice as long as intended.

```
keyGenerator, err :=
   scheme.Keygen(paillier.WithEachPrimeBitLen(lp.PaillierBitSizeN))
```

Figure B.11: *`bron-crypto/pkg/threshold/tsig/tecdsa/lindell17/keygen/trusted_dealer/trusted_dealer.go#L45`*

- **BLS fast signature verification follows a slow path.** In certain scenarios, BLS signature verification can be optimized. When these conditions are met, the library runs fast verification, but it does not return on success, so it repeats verification using the slower method.

```
if canRunFastAggregateVerify {
      ...
      if err := coreVerify(...); err != nil {
            return errs.WrapVerification(err, "could not verify fast
   aggregate signature")
      }
}
...
if err := coreAggregateVerify(...); err != nil {
      return errs.WrapVerification(err, "could not verify aggregate
   signature")
}
return nil
```

Figure B.12: *`bron-crypto/pkg/signatures/bls/participants.go#L432–L447`*

- **BLS signing error messages are misleading.** The following code switches on one value, but if it is unrecognized, the error string includes a different value. The error messages are incorrect.

```
switch v.rogueKeyAlg {
case Basic:
      ...
case POP:
      ...
case MessageAugmentation:
      ...
default:
      return errs.NewType("rogue key prevention scheme %d is not supported",
v.variant)
}
```

Figure B.13: `bron-crypto/pkg/signatures/bls/participants.go#L387–L424`

```
switch c.targetRogueKeyAlg {
case bls.Basic:
case bls.MessageAugmentation:
       ...
case bls.POP:
       ...
default:
       return nil, errs.NewType("unsupported rogue key prevention algorithm:
%d", c.scheme.RogueKeyPreventionAlgorithm())
}
```

Figure B.14: `bron-crypto/pkg/threshold/tsig/tbls/boldyreva02/signing/participant.go#L177–L197`

C. Automated Analysis Tool Configuration

As part of this assessment, we used the tools described below to perform automated testing of the codebase.

C.1. CodeQL

We used CodeQL to detect known vulnerabilities in the codebase. This analysis did not identify any issues.

```
# Create the Go database
codeql database create codeql.db --language=go --command="make build"

# Run queries
codeql database analyze codeql.db --format=sarif-latest --output=codeql.sarif \
  -- <QUERY PACK>
```

Figure C.1: The commands used to run CodeQL on the Bron Labs codebase

During the engagement, we ran the following query packs and query suites on the codebase:

- The `trailofbits/go-queries` query pack
- The `codeql/go-queries` query pack (included with CodeQL)
- The Go `security-extended` query suite (included with CodeQL)

We also ran a number of internally maintained CodeQL query suites on the codebase. This analysis identified a large number of unchecked type casts across the codebase that could potentially trigger panics if the corresponding input is not checked before the function is called.

C.2. go-mod-outdated

We used the `go-mod-outdated` tool to find any outdated dependencies of the codebase. It identified a few packages with newer versions than those used in the codebase, but nothing that constituted a security risk to the system.

```
go list -u -m -json all | go-mod-outdated
```

Figure C.2: The command used to run `go-mod-outdated` on the Bron Labs codebase

C.3. go-test

We used the built-in test coverage tool that ships with the Go compiler to generate test coverage metrics for the codebase.

```
make cover
```

Figure C.3: The command used to generate test coverage data for the codebase

We found that test coverage could be improved across the entire codebase. Some functions are not exercised by tests at all, and all components would benefit from more negative testing.

C.4. golangci-lint

We used the `golangci-lint` meta-linter that can be used to run a large number of linters and static-analysis tools on the codebase. This tool found many instances of unidiomatic code patterns in the codebase. Because of time constraints we did not go through all of them, but we still recommend the Bron Labs team to go through and address findings from `golangci-lint`.

```
golangci-lint run ./...
```

Figure C.4: The command used to run `golangci-lint` on the Bron Labs codebase

C.5. Nancy

We used the Nancy tool from Sonartype to identify known vulnerabilities in the codebase's Go dependencies. The tool did not identify any dependencies with known vulnerabilities.

```
go list -json -m all | nancy sleuth
```

Figure C.5: The command used to run Nancy on the Bron Labs codebase

C.6. Semgrep

We ran the static analyzer Semgrep with the rulesets shown in figure C.6 to identify low-complexity vulnerabilities in the codebase. These runs did not identify any issues or code quality findings.

```
git clone git@github.com:dgryski/semgrep-go.git

semgrep --metrics=off --sarif --config=auto
semgrep --metrics=off --sarif --config p/trailofbits
Semgrep --metrics=off --sarif --config /path/to/trailofbits/internal/semgrep/rules
```

Figure C.6: The command and rulesets used to run Semgrep on the Bron Labs codebase

We also wrote two custom Semgrep rules to perform variant analysis on the underlying issues in findings TOB-BRON-10 and TOB-BRON-20. This analysis of TOB-BRON-10 did not identify any other vulnerable locations in the codebase, but the analysis of TOB-BRON-20 found two additional implementations that contained the same vulnerability. We include the Semgrep rules used below.

```
rules:
  - id: unsafe-copy-operations-with-untrusted-bounds
    pattern-either:
      # Pattern 1: copy(target[len(target)-len(source):], source)
      - patterns:
          - pattern: copy($TARGET[len($TARGET) - len($SOURCE):], $SOURCE)
          - pattern-not-inside: |
              if len($SOURCE) <= len($TARGET) {
                ...
              }
          - pattern-not-inside: |
              if len($TARGET) >= len($SOURCE) {
                ...
              }

      # Pattern 2: copy(target[targetLen-sourceLen:], source)
      - patterns:
          - pattern: copy($TARGET[$EXPR:], $SOURCE)
          - metavariable-pattern:
              metavariable: $EXPR
              patterns:
                - pattern-either:
                    - pattern: $VAR1 - $VAR2
                    - pattern: len($VAR1) - len($VAR2)
                    - pattern: len($TARGET) - len($UNTRUSTED)
          - pattern-not-inside: |
              if $EXPR >= 0 {
                ...
              }

      # Pattern 3: Variable assignment then copy
      - patterns:
          - pattern-either:
              - pattern: |
                  $IDX := len($TARGET) - len($SOURCE)
                  ...
                  copy($TARGET[$IDX:], $SOURCE)
              - pattern: |
                  $IDX = len($TARGET) - len($SOURCE)
                  ...
                  copy($TARGET[$IDX:], $SOURCE)
          - pattern-not-inside: |
              if len($SOURCE) <= len($TARGET) {
                ...
              }

    message: |
      Unsafe copy operation detected. This pattern can cause a runtime panic if the
      source length exceeds the target length, resulting in a negative slice index.
    languages: [go]
    severity: ERROR
```

Figure C.7: Semgrep rule to detect unsafe copy operations

```
rules:
  - id: incomplete-map-equality-check
    patterns:
      # Match functions named Equal or similar
      - pattern-either:
          - pattern: |
              func ($RECV *$TYPE) Equal($OTHER *$TYPE) bool {
                ...
                for $KEY, $VAL := range $RECV.$MAPFIELD.Iter() {
                  ...
                  $OTHERVAL, $EXISTS := $OTHER.$MAPFIELD.Get($KEY)
                  ...
                }
                ...
              }
          - pattern: |
              func ($RECV *$TYPE) Equal($OTHER *$TYPE) bool {
                ...
                for $KEY, $VAL := range $RECV.$MAPFIELD.Iter() {
                  ...
                  $OTHERVAL := $OTHER.$MAPFIELD[$KEY]
                  ...
                }
                ...
              }
      # Ensure there's no length/size check before the iteration
      - pattern-not: |
          func ($RECV *$TYPE) Equal($OTHER *$TYPE) bool {
            ...
            if $RECV.$MAPFIELD.Size() != $OTHER.$MAPFIELD.Size() {
              ...
            }
            ...
            for $KEY, $VAL := range $RECV.$MAPFIELD.Iter() {
              ...
            }
            ...
          }
      - pattern-not: |
          func ($RECV *$TYPE) Equal($OTHER *$TYPE) bool {
            ...
            if len($RECV.$MAPFIELD) != len($OTHER.$MAPFIELD) {
              ...
            }
            ...
            for $KEY, $VAL := range $RECV.$MAPFIELD {
              ...
            }
            ...
          }
      # Also check for reverse iteration
    - pattern-not: |
        func ($RECV *$TYPE) Equal($OTHER *$TYPE) bool {
          ...
          for $KEY, $VAL := range $RECV.$MAPFIELD.Iter() {
            ...
          }
          ...
          for $KEY2, $VAL2 := range $OTHER.$MAPFIELD.Iter() {
            ...
          }
          ...
        }
    - pattern-not: |
        func ($RECV *$TYPE) Equal($OTHER *$TYPE) bool {
          ...
          for $KEY, $VAL := range $RECV.$MAPFIELD {
            ...
          }
          ...
          for $KEY2, $VAL2 := range $OTHER.$MAPFIELD {
            ...
          }
          ...
        }
  message: >-
    Potential incomplete map equality check in Equal() method.
  languages:
    - go
  severity: ERROR
```

Figure C.8: Semgrep rule to detect incomplete map equality checks

D. Constant-Time Constructions

Scoping for this project noted that constant-time analysis should be performed on a best-effort basis and that the entire codebase was not expected to be constant-time. We are sharing the following observations in that spirit. We do not consider these observations to rise to the level of security issues, but they are worth noting and considering as development moves forward.

Generalized multiplication leads to unnecessary complexity. The windowed multiplication technique in `pkg/base/algebra/impl/mul.go` relies on two operations: doubling and adding. These operations are used to build a lookup table of 16 elements, and appropriate small multiples of the multiplied group element are added to the product as needed. Assuming that the operations are constant-time, building the table should be a constant-time operation. However, we believe that, while the table is small and the items are likely to be 64-bit aligned, there is still a risk of cache-timing vulnerabilities.

Additionally, to be compatible with the interface, all elliptic curve implementations must provide separate `Double` and `Add` functions. When using complete formulas, the same routine can be used for both doubling and adding, obviating several timing side-channels. Despite using complete formulas for twisted Edwards and Weierstrass curves, both implementations provide separate `Double` and `Add` methods. While there are some optimizations available when doubling a point, this imposes a complexity penalty. Both functions must now be maintained separately, kept in sync over time as bugs are fixed, and be subjected to separate tests. The optimizations also invite future timing attacks if the functions are not used with sufficient care. The best way to deal with this is to turn the `Double` function into a thin wrapper around the `Add` function wherever possible; the overall performance degradation to the system overall will be minimal.

Constant-time module is superfluous. The constant-time byte, integer, slice, comparison, conditional load, and conditional swap functions in the `ct` module provide important functionality. However, the same functionality is also available in the Go standard library's `crypto.subtle` module. Adding a new type, `Choice`, increases the complexity of the library unnecessarily. We recommend switching to the `crypto.subtle` module.

Base58 encoding and decoding use data-dependent lookups. Base58 encoding and decoding use data-dependent indexing into the `Alphabet` and `b58` tables. As a byte table (rather than a 64-bit value table), lookups may be variable-time, even in cache. Some AMD and Intel chips, for instance, impose a penalty for loading any bytes that are not 32- or 64-bit aligned. Additionally, the decoder returns early when an invalid character is encountered; if an attacker can manipulate individual bits in a Base58-encoded value (say, by flipping bits in a Base58 string encrypted with AES-CTR), the early-return timing can leak information about the underlying plaintext. We are not aware of a Go library that performs Base58 encoding and decoding in a way that avoids table lookups, but we do recommend trying to avoid early decoding aborts by flagging invalid strings and returning an error after all characters have been decoded. Improved error checking would also prevent decode errors from being indistinguishable from successfully decoded empty strings.

E. POSEIDON Standards Compliance

Bron implements a variant of the POSEIDON hash function. While the parameter sets in the Bron implementation differ significantly from those discussed in the original paper, it is otherwise faithful to the structure and specification of the hash function. Further, the parameter sets used in the Bron implementation do not reduce the overall security level.

Specification Overview

The POSEIDON hash is a sponge construction that applies a permutation function to an internal state as new words (i.e., elements of a finite field) are incorporated via simple addition. The size `t` of the internal state is `t = r + c`, where `r` is the rate (the number of input words handled between permutations) and `c` is the capacity (the hidden portion of the state).

The permutation function consists of multiple *rounds*, each of which can be a “partial” round or a “full round.”

All rounds consist of three steps:

- `AddRoundConstants`, in which fixed values are added to each word of the internal state
- `SubWords`, in which words are substituted via exponentiation with exponent `e`
- `MixLayer`, in which the internal state is treated as a vector and multiplied by a maximum distance separable matrix

The key difference between partial rounds and full rounds is that, in full rounds, *every* word in the internal state undergoes a substitution. In partial rounds, only a single word undergoes a substitution.

The permutation function is parameterized in terms of the number of full rounds `f` and the number of partial rounds `p`. At the beginning of the permutation, the full round function is applied `f` times. The result is then fed into `p` iterations of the partial round function. Finally, the full round function is applied `f` times again.

This full-partial-full construction is intended to keep the algebraic degree of the hash function high while reducing the number of field operations, which is useful in the context of zkSNARK systems. The other tool for tweaking the degree of the hash function is the exponent `e` in the `SubWords` step, but increasing the exponent also increases the number of field operations.

One of the POSEIDON paper’s parameter sets that targets a 128-bit security level sets an exponent of `e = 5`, a full round count of `f = 8` (for a total of 16 full rounds), and a partial round count of `p = 60`. The algebraic degree of the resulting permutation is expected to be e^{2f+p} or 5^{76}.

Deviations from the Specification

The Bron POSEIDON implementation exhibits three small deviations from the POSEIDON specification. The largest difference comes from the decision to implement only full rounds. The other differences deal with parameterization and order of operations with initial round constants.

Full Rounds

The Bron POSEIDON implementation does not include support for partial rounds. Instead, *all* rounds are processed as full rounds. The POSEIDON paper makes it clear that partial rounds primarily serve to increase the algebraic degree of the permutation function to protect against algebraic attacks. As long as the total number of full rounds is increased to compensate for the lack of partial rounds, there should be no negative impact on security.

Parameter Sets

The parameter set used by Bron comes from the Kimchi proof system used by o1 Labs. The Kimchi parameters set `e = 7`, `f' = 55`, and `p = 0`. Note that `f'` is distinct from `f`, in that `f'` full rounds are applied only once.

The resulting algebraic degree is expected to be $2^{f'}$, or 7^{55}. This is somewhat lower than the algebraic degree provided by the POSEIDON paper's parameter set, but it is worth noting that both parameter sets satisfy the minimum algebraic degree criteria set in appendix C.2.1 of the POSEIDON paper.

We consider the Kimchi parameter set to be sufficient for Bron's stated security goals.

Operation Order and Initial Round Constants

The POSEIDON paper specifies an initial state of all zeroes, allowing the `AddRoundConstants` function to update the initial state according to the implementer's desires. However, there is no loss of security from using the all-zero initial state without the first round of constant addition.

This is the approach used in the Kimchi system, which serves as a reference for the Bron implementation. The Kimchi system also adds a final `AddRoundConstants` step to the *end* of the permutation. This effectively shifts the order of the operations from `AddRoundConstants -> SubWords -> MixLayer` to `SubWords -> MixLayer -> AddRoundConstants`. We are not aware of any security risk posed by this shift, though the final `AddRoundConstants` provides no value during a squeeze operation.

The Bron implementation supports this shifted approach, but also includes support for an initial set of round constants, which has the effect of shifting the order of operations back to match the POSEIDON paper.

F. Rogue-Key Attack Implementation

The following shows a working implementation of the rogue-key attack on the Gennaro DKG, as described in finding TOB-BRON-13.

```
package gennaro_test

import (
	"crypto/sha3"
	"io"
	"reflect"
	"testing"

	"github.com/stretchr/testify/require"

	"github.com/bronlabs/bron-crypto/pkg/base/curves/k256"
	"github.com/bronlabs/bron-crypto/pkg/base/datastructures/hashmap"
	"github.com/bronlabs/bron-crypto/pkg/base/datastructures/hashset"
	"github.com/bronlabs/bron-crypto/pkg/base/polynomials"
	"github.com/bronlabs/bron-crypto/pkg/base/prng/pcg"
	"github.com/bronlabs/bron-crypto/pkg/network"
	"github.com/bronlabs/bron-crypto/pkg/threshold/dkg/gennaro"
	"github.com/bronlabs/bron-crypto/pkg/threshold/sharing"
	"github.com/bronlabs/bron-crypto/pkg/threshold/sharing/feldman"
	"github.com/bronlabs/bron-crypto/pkg/threshold/sharing/shamir"
	"github.com/bronlabs/bron-crypto/pkg/transcripts"
	"github.com/bronlabs/bron-crypto/pkg/transcripts/hagrid"
)

// MaliciousParticipant wraps a Gennaro participant to execute the rogue key attack.
// Round1 and Round2 input handling behave identically to gennaro.Participant, but
// Round2 crafts a malicious Feldman vector that differs from the Pedersen
commitment.
type MaliciousParticipant struct {
	participant *gennaro.Participant[*k256.Point, *k256.Scalar]
	group       gennaro.Group[*k256.Point, *k256.Scalar]
	scalarField *k256.ScalarField
}
```

```
func NewMaliciousParticipant(
	sid network.SID,
	group gennaro.Group[*k256.Point, *k256.Scalar],
	id sharing.ID,
	ac *shamir.AccessStructure,
	tape transcripts.Transcript,
	prng io.Reader,
) (*MaliciousParticipant, error) {
	participant, err := gennaro.NewParticipant(sid, group, id, ac, tape, prng)
	if err != nil {
		return nil, err
	}
	return &MaliciousParticipant{
		participant: participant,
		group:       group,
		scalarField: k256.NewScalarField(),
	}, nil
}

func (m *MaliciousParticipant) SharingID() sharing.ID {
	return m.participant.SharingID()
}

func (m *MaliciousParticipant) Round1() (*gennaro.Round1Broadcast[*k256.Point,
*k256.Scalar], error) {
	return m.participant.Round1()
}

// Round2 executes the rogue key attack with a target public key contribution.
// The function uses Lagrange interpolation in the exponent to craft a Feldman
// vector whose constant term equals targetPoint.
func (m *MaliciousParticipant) Round2(
	t *testing.T,
	r1Broadcasts network.RoundMessages[*gennaro.Round1Broadcast[*k256.Point,
*k256.Scalar]],
	targetPoint *k256.Point,
	honestID sharing.ID,
) (*gennaro.Round2Broadcast[*k256.Point, *k256.Scalar],
network.RoundMessages[*gennaro.Round2Unicast[*k256.Point, *k256.Scalar]], error) {

	// Execute normal Round2 to get valid shares (from f'_1)
	_, round2Unicasts, err := m.participant.Round2(r1Broadcasts)
	if err != nil {
		return nil, nil, err
	}
	// Craft malicious Feldman vector with targetPoint as constant term
	rogueFeldmanVector := createRogueFeldmanVector(
		t,
		m.group,
		m.scalarField,
		targetPoint,
		honestID,
		round2Unicasts,
	)

	round2Broadcast := &gennaro.Round2Broadcast[*k256.Point, *k256.Scalar]{
		FeldmanVerificationVector: rogueFeldmanVector,
	}

	return round2Broadcast, round2Unicasts, nil
}

// createRogueFeldmanVector constructs a Feldman vector with targetPoint as constant
term.
func createRogueFeldmanVector(
	t *testing.T,
	group gennaro.Group[*k256.Point, *k256.Scalar],
	scalarField *k256.ScalarField,
	targetPoint *k256.Point,
	honestID sharing.ID,
	shares network.RoundMessages[*gennaro.Round2Unicast[*k256.Point,
*k256.Scalar]],
) feldman.VerificationVector[*k256.Point, *k256.Scalar] {

	honestShare, exists := shares.Get(honestID)
	require.True(t, exists)
	honestShareValue := honestShare.Share.Value()
	honestSharePoint := group.Generator().ScalarMul(honestShareValue)

	honestScalar := shamir.SharingIDToLagrangeNode(scalarField, honestID)
	zero := scalarField.Zero()

	// Construct P(x) = targetPoint + x * linearCoefficient where P(honestScalar)
= honestSharePoint
	deltaPoint := honestSharePoint.Sub(targetPoint)
	invHonestScalar, err := honestScalar.TryInv()
	require.NoError(t, err)
	linearTerm := deltaPoint.ScalarMul(invHonestScalar)

	// Construct Feldman vector using reflection
	rogueFeldmanVector := &polynomials.ModuleValuedPolynomial[*k256.Point,
*k256.Scalar]{}
	coeffsField :=
reflect.ValueOf(rogueFeldmanVector).Elem().FieldByName("coeffs")
	coeffsField = reflect.NewAt(coeffsField.Type(),
coeffsField.Addr().UnsafePointer()).Elem()
	coeffsField.Set(reflect.ValueOf([]*k256.Point{targetPoint, linearTerm}))

	// Verify interpolation
	require.True(t,
rogueFeldmanVector.Eval(honestScalar).Equal(honestSharePoint))
	require.True(t, rogueFeldmanVector.Eval(zero).Equal(targetPoint))

	return rogueFeldmanVector
}

// TestRogueKeyAttack demonstrates forcing the DKG to generate a specific public
// key.
func TestRogueKeyAttack(t *testing.T) {
	threshold := uint(3)
	group := k256.NewCurve()
	scalarField := k256.NewScalarField()
	sid := network.SID(sha3.Sum256([]byte("rogue-key-attack")))
	tape := hagrid.NewTranscript("rogue-key-attack")
	prng := pcg.NewRandomised()

	// Create 1 honest and 2 malicious participants.
	honestID := sharing.ID(1)
	maliciousIDs := []sharing.ID{2, 3}
	allIDs := []sharing.ID{1, 2, 3}

	shareholders := hashset.NewComparable[sharing.ID]()
	for _, id := range allIDs {
		shareholders.Add(id)
	}
	ac, err := shamir.NewAccessStructure(threshold, shareholders.Freeze())
	require.NoError(t, err)

	honestParticipant, err := gennaro.NewParticipant(sid, group, honestID, ac,
tape.Clone(), prng)
	require.NoError(t, err)

	maliciousParticipants := make([]*MaliciousParticipant, len(maliciousIDs))
	for i, id := range maliciousIDs {
		maliciousParticipants[i], err = NewMaliciousParticipant(sid, group, id,
ac, tape.Clone(), prng)
		require.NoError(t, err)
	}

	// Round 1: All participants follow the protocol
	round1Broadcasts := make(map[sharing.ID]*gennaro.Round1Broadcast[*k256.Point,
*k256.Scalar])
	round1Broadcasts[honestID], err = honestParticipant.Round1()
	require.NoError(t, err)

	for i, id := range maliciousIDs {
		round1Broadcasts[id], err = maliciousParticipants[i].Round1()
		require.NoError(t, err)
	}

	// Round 2: Honest participant executes normally
	round2Input := hashmap.NewComparable[sharing.ID,
*gennaro.Round1Broadcast[*k256.Point, *k256.Scalar]]()
	for _, id := range allIDs {
		if id != honestID {
			round2Input.Put(id, round1Broadcasts[id])
		}
	}

	honestR2Broadcast, _, err := honestParticipant.Round2(round2Input.Freeze())
	require.NoError(t, err)
	honestContribution :=
honestR2Broadcast.FeldmanVerificationVector.Eval(scalarField.Zero())
	t.Logf("Honest contribution: %v", honestContribution)

	// Attackers choose target secret and compute required Feldman constant
terms:
	// - Malicious participant 2 chooses: identity
	// - Malicious participant 3 chooses: targetScalar * G - honestContribution
	targetScalar, err := scalarField.Random(prng)
	require.NoError(t, err)

	targetPublicKey := group.Generator().ScalarMul(targetScalar)
	maliciousContribution2 := group.OpIdentity()
	maliciousContribution3 := targetPublicKey.Sub(honestContribution)

	// Round 2: Malicious participants craft Feldman vectors with target constant
terms
	maliciousRound2Broadcasts :=
make(map[sharing.ID]*gennaro.Round2Broadcast[*k256.Point, *k256.Scalar])
	maliciousRound2Unicasts :=
make(map[sharing.ID]network.RoundMessages[*gennaro.Round2Unicast[*k256.Point,
*k256.Scalar]])
```

```
	targetPoints := []*k256.Point{maliciousContribution2, maliciousContribution3}

	for i, id := range maliciousIDs {
		round2InputMalicious := hashmap.NewComparable[sharing.ID,
*gennaro.Round1Broadcast[*k256.Point, *k256.Scalar]]()
		for _, otherId := range allIDs {
			if otherId != id {
				round2InputMalicious.Put(otherId,
round1Broadcasts[otherId])
			}
		}

		maliciousRound2Broadcasts[id], maliciousRound2Unicasts[id], err =
			maliciousParticipants[i].Round2(t,
round2InputMalicious.Freeze(), targetPoints[i], honestID)
		require.NoError(t, err)
	}

	// Round 3: Honest participant accepts malicious inputs
	round3BroadcastInput := hashmap.NewComparable[sharing.ID,
*gennaro.Round2Broadcast[*k256.Point, *k256.Scalar]]()
	round3UnicastInput := hashmap.NewComparable[sharing.ID,
*gennaro.Round2Unicast[*k256.Point, *k256.Scalar]]()

	for _, id := range maliciousIDs {
		round3BroadcastInput.Put(id, maliciousRound2Broadcasts[id])
		unicastToHonest, exists := maliciousRound2Unicasts[id].Get(honestID)
		require.True(t, exists)
		round3UnicastInput.Put(id, unicastToHonest)
	}

	honestOutput, err := honestParticipant.Round3(round3BroadcastInput.Freeze(),
round3UnicastInput.Freeze())
	require.NoError(t, err)
	require.NotNil(t, honestOutput)

	// Verify attack: final PK = targetPublicKey
	publicKey := honestOutput.PublicKeyValue()
	require.True(t, publicKey.Equal(targetPublicKey), "final PK should equal
target PK")
}
```

Figure E.1: Implementation of a rogue-key attack on the Gennaro DKG in a 3-out-of-3 setting

G. 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.

From December 17 to December 19, 2025, Trail of Bits reviewed the fixes and mitigations implemented by the Bron Labs team for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue.

We reviewed a number of commits across three different pull requests. For the issue related to test coverage (TOB-BRON-25), we recomputed the test coverage using the latest commit on the master branch at the time of the fix review.

In summary, of the 25 issues described in this report, Bron Labs has resolved 24 issues and has partially resolved the remaining issue. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Severity	Status
1	Signature aggregation rejects valid recovery IDs	Informational	Resolved
2	The base OT receivers use the same choice bits for all senders in DKLs23	Undetermined	Resolved
3	VSOT with naive batching does not provide endemic security	Undetermined	Resolved
4	VSOT sender inconsistently rejects identity point	Informational	Resolved
5	SoftSpokenOT does not include all row indices in the final hash	Undetermined	Resolved
6	OT extension PRG is not necessarily unique due to potential seed collision	Informational	Resolved
7	BBOT output conversion can include additional domain separation	Informational	Resolved
8	DKLs23 consistency check failures have wrong error type	Undetermined	Resolved

9	Fischlin proofs are not bound to the prover	Low	Resolved
10	The Fischlin verifier panics on malicious inputs	Low	Resolved
11	The library does not reject zero sharing IDs	Informational	Resolved
12	Malicious participant can increase sharing threshold for Feldman and Pedersen VSS	Medium	Resolved
13	Gennaro DKG is vulnerable to rogue-key attacks by a malicious participant	High	Resolved
14	Paillier nonces are sampled from the wrong space	Informational	Resolved
15	The LPDL verifier accepts forged proofs	Low	Resolved
16	The library generates zero-sharing IDs	Informational	Resolved
17	BLS key generation does not match standard recommendations	Informational	Resolved
18	The pailliern proof does not incorporate statement	Informational	Resolved
19	Paillier range proof serialization problems	Low	Resolved
20	Public key material equality test does not ensure equality	Informational	Resolved
21	Inconsistent Paillier plaintext representations	Informational	Resolved
22	Ineffective input validation in LPDL verifier	Informational	Resolved
23	Malicious Alice can use invalid Paillier ciphertext to recover Lindell17 key	High	Resolved
24	Paillier range proof verifier fails to check input lengths	Low	Resolved

25	Insufficient test coverage	Low	**Partially Resolved**

Detailed Fix Review Results

TOB-BRON-1: Signature aggregation rejects valid recovery IDs
Resolved in commit `7d2addc`. Signature aggregation now accepts all valid recovery IDs.

TOB-BRON-2: The base OT receivers use the same choice bits for all senders in DKLs23
Resolved in commit `e89f75f`. The choice generation is moved inside the loop such that each OT instance gets its own choice bits.

TOB-BRON-3: VSOT with naive batching does not provide endemic security
Resolved in commit `00dc082`. The hash input now includes the batch index, which breaks correlations between related or equal curve input points.

TOB-BRON-4: VSOT sender inconsistently rejects identity point
Resolved in commit `f3cfdb0`. The VSOT sender no longer rejects the identity point.

TOB-BRON-5: SoftSpokenOT does not include all row indices in the final hash
Resolved in commits `fc573d3` and `9ae6ef9`. Both SoftSpokenOT participants now include the indices of the batch and the row within the batch in the final hash.

TOB-BRON-6: OT extension PRG is not necessarily unique due to potential seed collision
Resolved in commit `9ae6ef9`. Both SoftSpokenOT participants now use a keyed Blake2b extensible-output function (XOF) as the PRG, where the session ID is the key.

TOB-BRON-7: BBOT output conversion can include additional domain separation
Resolved in commit `24d5235`. BBOT output conversion now has the following additional domain separation: both participants key the Blake2b hash function using values extracted from the transcript, including a randomizer label and the labeled IDs of sender and receiver. For each message, the hash now also includes the indices `xi` and `l`, such that equal messages will not result in equal output.

TOB-BRON-8: DKLs23 consistency check failures have wrong error type
Resolved in commits `06dbf87`, `ada9502`, and `fdb6108`. The consistency check now generates an error of type `IdentifiableAbort`. The consistency check for the public key also generates an error of type `TotalAbort`.

TOB-BRON-9: Fischlin proofs are not bound to the prover
Resolved in commits `9f5593d` and `fc04249`. The Fischlin compiler now includes a new key extracted from the transcript. This means that each caller must add the prover identity to the transcript when constructing a new compiler. As of the latter commit, all callers (Lindell17 DKG and signing, Lindell22 signing, Gennaro DKG, and DKLs23 DKG) satisfy this requirement.

TOB-BRON-10: The Fischlin verifier panics on malicious inputs
Resolved in commit `6c40f93`. The Fischlin verifier checks the length of the provided challenge and returns an error if it is invalid.

TOB-BRON-11: The library does not reject zero sharing IDs
Resolved in commit `82ce433`. Constructing an access structure containing zero will now return an error.

TOB-BRON-12: Malicious participant can increase sharing threshold for Feldman and Pedersen VSS
Resolved in commits `9285661` and `06dbf87`. The Feldman and Pedersen verifiers now check that the degree matches the threshold.

TOB-BRON-13: Gennaro DKG is vulnerable to rogue-key attacks by a malicious participant
Resolved in commit `4a6a161`. Each participant now provides a batch Schnorr proof on the coefficients of their Feldman verification vector. This prevents a rogue-key attack because malicious participants cannot know the discrete logarithm of their target constant term, as it depends on the secret constant terms of the honest participants. Note that rushing adversaries can still bias the resulting public key outside of the honest-majority setting, because the Pedersen commitment does not bind to the Feldman verification vector in this case. However, this is not sufficient to perform a rogue-key attack, and there are no known attacks relying on this potential bias.

TOB-BRON-14: Paillier nonces are sampled from the wrong space
Resolved in commit `d6799e7`. Nonces are now defined inside the RSA group rather than the Paillier group, and the methods for generation and creation of nonces from natural numbers verify that the nonce is coprime to the modulus.

TOB-BRON-15: The LPDL verifier accepts forged proofs
Resolved in commit `29e50fa`. The verifier now generates the b parameter from $\mathbb{Z}/q^2\mathbb{Z}$ as required.

TOB-BRON-16: The library generates zero-sharing IDs
Resolved in commit `4dfa2c7`. Share generation now uses index `i + 1` to ensure that no share is zero.

TOB-BRON-17: BLS key generation does not match standard recommendations
Resolved in commit `1319424`. The key generation now follows the recommendation from the specification.

TOB-BRON-18: The pailliern proof does not incorporate statement
Resolved in commit `06dbf87`. The proof now incorporates the statement by including the modulus.

TOB-BRON-19: Paillier range proof serialization problems
Resolved in commit 06dbf87. The Bytes methods of Statement, Commitment, and Response structs now prepend the length of each component as an unsigned 64-bit value. The Serialize* functions have been removed.

TOB-BRON-20: Public key material equality test does not ensure equality
Resolved in commits 06dbf87 and ada9502. For BasePublicMaterial, PublicMaterial, and AuxiliaryInfo, the equality check now also compares sizes of the contained structures such that inclusion becomes equivalent to equality because they have the same size.

TOB-BRON-21: Inconsistent Paillier plaintext representations
Resolved in commit 40c2d9d. The plaintext validation function now checks whether the plaintext is in the correct symmetric range.

TOB-BRON-22: Ineffective input validation in LPDL verifier
Resolved in commit 5b58a46. The input validation no longer checks the encrypted value, except that it is not nil.

TOB-BRON-23: Malicious Alice can use invalid Paillier ciphertext to recover Lindell17 key
Resolved in commit c2a6aad. Constructing Paillier nonces and ciphertexts now includes a check that the element is coprime to the modulus. All methods in the underlying znstar module that construct elements verify whether these elements are coprime to the modulus. The corresponding routines for unmarshalling received data use these methods.

TOB-BRON-24: Paillier range proof verifier fails to check input lengths
Resolved in commit 06dbf87. The verifier now checks the lengths of all commitment and response arrays.

TOB-BRON-25: Insufficient test coverage
Partially resolved as of commit afbba54. Several modules including the Fiat-Shamir compiler now have test coverage. However, there are still modules that do not have test coverage, including the polynomial implementation used for secret sharing and interpolation.

H. 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.

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review assessments, supporting client organizations in the technology, defense, blockchain, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, Uniswap, Solana, Ethereum Foundation, Linux Foundation, and Zoom.

To keep up with our latest news and announcements, please follow @trailofbits on X or LinkedIn and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact or email us at info@trailofbits.com.

Trail of Bits, Inc.
228 Park Ave S #80688
New York, NY 10003
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

Trail of Bits performed all activities associated with this project in accordance with a statement of work and an agreed-upon project plan.

Security assessment projects are time-boxed and often rely on information 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 software controls and security properties. 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. A project's time and resource constraints also limit their use.