AnySwap Threshold-DSA

Security Assessment

February 22, 2022

Prepared For:
Zhengxin Gao | *AnySwap*
zhengxin.gao@anyswap.exchange

Prepared By:
Filipe Casal | *Trail of Bits*
filipe.casal@trailofbits.com

Jim Miller | *Trail of Bits*
james.miller@trailofbits.com

Changelog:

November 10, 2021:	Initial report draft
February 22, 2022:	Added Appendix D: Fix Log

Executive Summary

From October 18 to November 8, 2021, AnySwap engaged Trail of Bits to review the security of its implementation of a threshold signature scheme. Trail of Bits originally conducted this assessment over six person-weeks, with two engineers working from commit `0d2d054b` of the [`FastMulThreshold-DSA` repository](). In the final week of the assessment, AnySwap provided updates in commit `036df7c9` for Trail of Bits to review.

We used the first week of the assessment to analyze the [GG20 paper](), which the implementation is based on, and AnySwap's documentation of its threshold signature scheme. From there, we began a preliminary review of the codebase, focusing primarily on `smpc-lib`. In addition to this, we used static analysis tooling, including [gosec](), [Staticcheck](), and [Semgrep](), to find vulnerable code patterns throughout the codebase.

In the second week of the assessment, we finished reviewing the `smpc-lib` portion of the codebase. From there, we began reviewing other components of the system, including `smpc` and the message transport layer.

In the final week, we completed our review of all of the components that were prioritized by AnySwap. We also reviewed the changes supplied by AnySwap in the updated commit hash `036df7c9`.

Our review resulted in 27 findings ranging from high to informational severity. The 12 high-severity issues stem from improper error handling and the improper implementation of certain zero-knowledge proof systems and verifiable secret sharing (VSS) schemes, which could leak secret data. The eight medium-severity issues pertain to missing input validation and incorrect implementations, which could compromise the security of the protocol. The two low-severity issues also stem from insufficient input validation. Lastly, the five informational-severity issues pertain to minor deviations from the protocol specification and unnecessary parameter generation.

The significant number of high-severity issues discovered during our review are indicative of an immature codebase that has room for improvement. These issues largely stem from incorrect protocol implementation and improper data validation. Several of these issues affect critical areas, and we suspect that similar issues are present elsewhere in the codebase. Therefore, we recommend that AnySwap focus on protocol implementation and data validation moving forward.

In addition to addressing the issues described in this report, we recommend that AnySwap take the following steps to improve the codebase moving forward. First, it is imperative that AnySwap improve the system's data validation, as several of the issues that we identified stem from this vulnerability class. AnySwap should review all of its critical functions and ensure that all of the inputs are validated properly. It may be beneficial to consolidate all the system's validation into a single location, such as a dedicated function, rather than performing several checks in an ad hoc fashion. In addition to this, we recommend that AnySwap reduce the system's large amount of code duplication and consolidate some functions into a single interface; the current implementation is complex, and these two qualities in particular make it easier to introduce mistakes.

Update: After the completion of the assessment, Trail of Bits reviewed fixes implemented for issues presented in this report. See the detailed fix log in Appendix D.

Project Dashboard

Application Summary

Name	AnySwap Threshold-DSA
Version	`FastMulThreshold-DSA`: `0d2d054b`, `036df7c9`
Type	Go
Platforms	Multiple

Engagement Summary

Dates	October 18–November 8, 2021
Method	Full knowledge
Consultants Engaged	2
Level of Effort	6 person-weeks

Vulnerability Summary

Total High-Severity Issues	12	■■■■■■■■■■■■
Total Medium-Severity Issues	8	■■■■■■■■
Total Low-Severity Issues	2	■■
Total Informational-Severity Issues	5	■■■■■
Total Undetermined-Severity Issues	0	
Total	27	

Category Breakdown

Cryptography	24	■■■■■■■■■■■■■■■■■■■■■■■■
Data Validation	3	■■■
Total	27	

Code Maturity Evaluation

Category Name	Description
Access Controls	**Weak.** The implementation currently lacks secure broadcast (TOB-ATSS-019) and peer-to-peer (TOB-ATSS-020) channels, which renders the protocol insecure. Additionally, user inputs in critical functions are not sufficiently validated in several locations.
Arithmetic	**Weak.** In multiple locations, traditional arithmetic is used in place of modular arithmetic, which leads to significant issues. Additionally, unsafe use of modular inverses could cause crashes in multiple locations.
Function Composition	**Moderate.** The threshold signature scheme implementation is cleanly separated into files for each phase of the protocol. However, the codebase contains a significant amount of duplicate code that could be consolidated into a dedicated function.
Key Management	**Weak.** AnySwap implements a complex, multistep threshold signature scheme with several checks in each step designed to protect the group's signing key. We identified several issues stemming from deviations from this protocol and a lack of cryptographic best practices.
Specification	**Moderate.** AnySwap provided high-level documentation for its implementation and a paper for its ECDSA implementation. However, there is no specification corresponding to the EdDSA implementation, and important high-level information is missing from the documentation (e.g., information on the transport layer). Moreover, various locations use short variable names with minimal surrounding comments.
Testing and Verification	**Moderate.** AnySwap has unit tests for certain critical components. However, certain components do not have any tests, and they lack advanced testing techniques, such as property testing.

Engagement Goals

The engagement was scoped to provide a security assessment of AnySwap's threshold signature scheme implemented in the `FastMulThreshold-DSA` repository.

Specifically, we sought to answer the following questions:

- Does the implementation of the threshold signature scheme in `FastMulThreshold-DSA` conform to its specifications?
- Could a malicious party cause an insecure signing key to be generated?
- Could a malicious party forge group signatures?
- Could a malicious party recover the secrets of other parties?
- Are either the ECDSA or EdDSA threshold signature schemes vulnerable to known cryptographic attacks?
- Does the implementation conform to cryptographic best practices?

Coverage

We performed a detailed manual review of both the `smpc-lib` and `smpc` portions of the codebase. `smpc-lib` implements both the ECDSA and EdDSA threshold signature schemes and the corresponding zero-knowledge proofs that they require. Our review focused on identifying deviations of the implementation from the specification and insufficient input validation for critical functions, as these tend to lead to severe issues. Additionally, we focused on ensuring that the implementation does not violate any assumptions, uses cryptographic best practices, and does not contain various low-level issues, such as missing error checks. In `smpc`, we focused on ensuring that all data and messages are parsed and validated properly and that the message transport layer was implemented securely. Lastly, we performed a high-level manual review of the `crypto`, `internal`, `cmd`, and `rpc/smpc` components, as prioritized by AnySwap. We also used gosec, Staticcheck, and Semgrep to assist our review for all components.

Recommendations Summary

This section aggregates all the recommendations made during the engagement. Short-term recommendations address the immediate causes of issues. Long-term recommendations pertain to the development process and long-term design goals.

Short term

❑ **Include the public value g^{sk} in the computation of the hash function.** TOB-ATSS-001

❑ **Add a check to guarantee that the `id` of the user for whom the protocol is generating a share is not zero modulo the curve of the protocol.** TOB-ATSS-002

❑ **In addition to logging the error, ensure that the protocol either exits the function or terminates the program with an error (depending on the case).** TOB-ATSS-003

❑ **Replace the condition with `if proof.S1.Cmp(s256.S256().N3()) > 0`.** TOB-ATSS-004

❑ **Add a check to verify that `(r, s)` is in the interval `[1, q-1]`, that the provided public key is on the correct elliptic curve, and that the public key is not the point at infinity.** TOB-ATSS-005

❑ **Change the comparison between h_1 and h_2 to be performed modulo Ñ.** Also, add checks to ensure that h_1 and h_2 are not equal to zero or one modulo Ñ. Finally, have the prover include a proof that Ñ is the product of two primes using the proof system from Section 3.4 of Goldberg et al. (2019). Ensure that the verifier verifies this proof as well. TOB-ATSS-006

❑ **Add checks in `Vss`, `Vss2`, and `CheckAddPeer` to ensure that `t` is greater than one and less than or equal to `n`.** TOB-ATSS-007

❑ **Pick a custom domain separator specific to AnySwap and this protocol.** Identify all instances in which a hash is computed over multiple inputs that do not have a fixed size, and add this domain separator to these instances. TOB-ATSS-008

❑ **Include the public statement values in the computation of the hash function.** TOB-ATSS-009

❑ **Include the discrete logarithm statement in the "MtA with check" proof according to the proof described in Appendix A.2 of the GG18 paper.** Notice the typo in the paper's verification description: where it reads "The Verifier checks that $s_1 \leq q^3$, $g^1 = X^e u \in G$," it should read "The Verifier checks that $s_1 \leq q^3$, $g^{s1} = X^e u \in G$." TOB-ATSS-010

❑ **Implement phases 5 and 6 as described in section 5.1 of the GG20 spec.** TOB-ATSS-011

❑ **Add a zero-knowledge proof that E_i is a square-free integer, such as the proof in section 3.1 of Gennaro et al.** (1998), or the proof in section 3.2 of Goldberg et al. (2019). Additionally, add a check to the `GenerateKeyPair` function to verify that the generated prime numbers are not equal. TOB-ATSS-012

❑ **Revise the implementation so that the Paillier keys and Fujisaki-Okamoto commitments $\tilde{N}$, h_1, and h_2 have to be generated only for new party members.** TOB-ATSS-013

❑ **Adjust the implementation of all three MtA proof systems to reduce `e` modulo `s256.S256().N`.** TOB-ATSS-014

❑ **Adjust the implementation of all three MtA proof verifiers to reconstruct the `N2` and `G` values rather than using the values submitted by the prover.** TOB-ATSS-015

❑ **Add the following checks to the MtA verification functions for the first proof system:** TOB-ATSS-016

❑ **Replace the Schnorr proof of knowledge of u_i with a proof of knowledge of x_i.** TOB-ATSS-017

❑ **Implement a mechanism to check that all incoming messages have the required fields and that all values are correctly parsed in `smpc/{key_ec.go, key_ed.go, sign_ed.go, sign_ec.go, reshare.go}`.** TOB-ATSS-018

❑ **Add a check to ensure that participants sign their broadcasted messages.** Additionally, ensure that when the system handles incoming messages, it ensures the validity of the signature and checks that the signature comes from the user in the `fromId` field in the message. TOB-ATSS-019

❑ **Add a check to ensure that participants sign their peer-to-peer messages before encrypting them.** Additionally, ensure that when the system handles incoming messages, it decrypts the message and ensures the validity of the signature by checking whether it comes from the user in the `fromId` field in the message. TOB-ATSS-020

❑ **Add a check in the implementation that verifies that `e` is not `nil`, and ensure that the system falls back if it is.** TOB-ATSS-021

❑ **Add a check to verify that `deltaSumInverse` is not `nil`, and ensure that the system falls back if it is.** TOB-ATSS-022

❑ **Check all uses of modular inverse functions and their return values for errors.** Additionally, include a verification step to guarantee that all participants have modularly different node IDs. TOB-ATSS-023

❑ **Adjust the implementation in round 7 so that each party returns an error if the EdDSA signature is not properly verified.** TOB-ATSS-024

❑ **Include the public values `S` and `R` in the computation of the hash function.** TOB-ATSS-025

❑ **Modify the system so that it creates a new generator in a verifiable way.** One way to do this is to hash the coordinates of the base generator with a counter until this hash is a valid x-coordinate for a point in the elliptic curve. TOB-ATSS-026

❑ **Validate all input from the proof statement to be modularly different from zero or one (in the case of a scalar) and different from the point at infinity (in the case of a point of an elliptic curve).** TOB-ATSS-027

Long term

❑ **Review all utilizations of the Fiat-Shamir transformation for missing public parameters and use a domain separator to separate each value.** TOB-ATSS-001, TOB-ATSS-009

❑ **Do not use user-provided parameters to generate the secret shares.** Instead, create the shares by evaluating the polynomial at x starting with 1, 2, ..., up to the number of participating players. TOB-ATSS-002

❑ **Incorporate static analysis tools such as errcheck, Semgrep, and CodeQL in the development process to find instances in which returned errors are not checked or acted upon.** TOB-ATSS-003

❑ **Review all zero-knowledge proof verifications to ensure they properly validate all input values.** TOB-ATSS-006

❑ **Review all functions for critical components of the protocol and ensure they have comprehensive input validation.** TOB-ATSS-007, TOB-ATSS-016

❑ **Add the custom domain separator to all hash computations.** TOB-ATSS-008

❑ **Document the details of both the broadcast and peer-to-peer channels.**
TOB-ATSS-019, TOB-ATSS-020

❑ **Review all utilizations of the Fiat-Shamir transformation for missing public parameters.** TOB-ATSS-025

Findings Summary

#	Title	Type	Severity
1	Weak Fiat-Shamir transformation in Schnorr's zero-knowledge proof	Cryptography	High
2	VSS generates shares from ID coordinates	Cryptography	High
3	Unhandled errors in cryptographically sensitive operations	Data Validation	High
4	MtA proof verification always rejects the upper bound q^3	Cryptography	Informational
5	ECDSA signature verification can panic	Cryptography	Medium
6	Missing checks in NtildeProof verification could lead to Golden Shoe attack	Cryptography	High
7	Insufficient input validation in Feldman VSS operations could result in trivial or unrecoverable shares	Cryptography	Low
8	Missing domain separation in various hash computations could result in canonicalization attacks	Cryptography	High
9	Weak Fiat-Shamir transformation in MtA proofs	Cryptography	High
10	Nonbinding "MtA with check" proof	Cryptography	Medium
11	Missing implementation of phase 5 of the signing protocol	Cryptography	Medium
12	Paillier modulus is not proved to be square-free	Cryptography	Medium
13	Unnecessary key generation in the reshare protocol	Cryptography	Informational
14	Fiat-Shamir challenge in MtA proofs use the wrong modulus	Cryptography	Informational
15	MtA proof verification implicitly trusts the prover's Paillier public key	Cryptography	High

16	Insufficient input validation in MtA verify functions could allow proof forgery	Cryptography	Low
17	Key generation includes the wrong Schnorr proof	Cryptography	Informational
18	Incoming messages are not validated and could cause crashes	Data Validation	Medium
19	In the transport layer, broadcast messages are not signed	Cryptography	High
20	In the transport layer, peer-to-peer messages are not signed	Cryptography	High
21	Queue InsertBefore implementation does not check for nil	Data Validation	Informational
22	Nonexistent modular inverse leads to crash	Cryptography	Medium
23	Modularly equal nodeIDs cause crashes when signing and resharing	Cryptography	Medium
24	EdDSA protocol ignores final signature verification	Cryptography	Medium
25	Weak Fiat-Shamir transformation in zero-knowledge proof of phase 6	Cryptography	High
26	Nonbinding Pedersen commitments	Cryptography	High
27	Insufficient input validation in the zero-knowledge proofs of phase 3 and phase 6	Cryptography	High

1. Weak Fiat-Shamir transformation in Schnorr's zero-knowledge proof

Severity: High	Difficulty: Low
Type: Cryptography	Finding ID: TOB-ATSS-001
Target: `smpc-lib/crypto/ed/schnorrZK.go`	

Description

The `smpc-lib/crypto/` folder contains implementations of several zero-knowledge proof systems that are required in the threshold signature protocol GG20. These zero-knowledge proof systems are essential to the security guarantees of the protocol, ensuring that the parties remain honest. The Schnorr zero-knowledge proof system is used in both the key generation phase and the signing phase of the protocol. It assures the verifier, who knows the public value g^{sk}, that the prover knows the discrete log value `sk`.

The implementation follows the Fiat-Shamir transformation and uses a hash function to non-interactively generate the verifier challenge `e`.

```
...
h := sha512.New()
h.Write(RBytes[:])
h.Write(message[:])
h.Sum(eDigest[:0])
...
```

Figure 1.1: `smpc-lib/crypto/ed/schnorrZK.go:50`

However, this transformation is known to be insecure if the calculation of the hash does not include the entire proof statement. Here, the calculation includes only the `Rbytes` value and a salt `message`, missing the public value that we are trying to prove, g^{sk}.

Exploit Scenario

An attacker notices that this proof system verification does not include the public key and is able to forge proofs, undermining the security guarantees of the protocol.

Recommendations

Short term, include the public value g^{sk} in the computation of the hash function.

Long term, review all utilizations of the Fiat-Shamir transformation for missing public parameters and use a domain separator to separate each value.

2. VSS generates shares from ID coordinates

Severity: High
Type: Cryptography
Target: `smpc-lib/crypto/ed/vss.go`

Difficulty: Low
Finding ID: TOB-ATSS-002

Description

The implementation of the threshold signature scheme uses Feldman's verifiable secret sharing (VSS) protocol to split secrets into shares, which are then distributed to other parties. The VSS protocol works by hiding the secret in the constant term of a random polynomial:

$$p(x) = \text{secret} + a_1x + \ldots + a_nx^n$$

Then, the secret is split into shares by providing the polynomial evaluated at specific points to each party member.

In the code, the shares are generated by evaluating the polynomial with the other parties' IDs. It is critical that these party IDs are non-zero because evaluating the polynomial at point 0 reveals the secret.

However, the code does not check that these IDs are nonzero, or zero modulo the order of the curve. Therefore, one can send the order of the elliptic curve as one's ID, which will evaluate to 0 when the polynomial evaluation modulo the curve order is performed.

```
for k, v := range uids {
	share := calculatePolynomial(cfs, v)
	shares[k] = share
}
```

Figure 2.1: `smpc-lib/crypto/ed/vss.go#L99-L102`

```
func (polyStruct *PolyStruct2) Vss2(ids []*big.Int) ([]*ShareStruct2, error) {

	shares := make([]*ShareStruct2, 0)

	for i := 0; i < len(ids); i++ {
		shareVal := calculatePolynomial2(polyStruct.Poly, ids[i])
		shareStruct := &ShareStruct2{ID: ids[i], Share: shareVal}
		shares = append(shares, shareStruct)
	}

	return shares, nil
}
```

Figure 2.2: `smpc-lib/crypto/ec2/vss2.go#L74-L86`

Thus, a party with an ID equal to zero or to the order of the curve will receive the secret as its share.

Exploit Scenario

An attacker configures his `id` as the order of the curve. When he obtains the shares from the other users, they will be equal to the other users' secret keys, allowing the attacker to sign messages as the other members.

Recommendations
Short term, add a check to guarantee that the `id` of the user for whom the protocol is generating a share is not zero modulo the curve of the protocol.

Long term, do not use user-provided parameters to generate the secret shares. Instead, create the shares by evaluating the polynomial at `x` starting with 1, 2, ..., up to the number of participating players.

3. Unhandled errors in cryptographically sensitive operations

Severity: High	Difficulty: High
Type: Data Validation	Finding ID: TOB-ATSS-003
Target: Various files	

Description

In several locations of the codebase, an error is detected and logged, but then the function continues as if the error did not occur. This may lead to undefined behavior or panics, or cause cryptographic material such as keys or the coefficients of the VSS polynomial to be left as zero.

```
if _, err := io.ReadFull(rand, rndNum[:]); err != nil {
	fmt.Println("Error: io.ReadFull(rand, rndNum[:])")
}
```

Figure 3.1: An error is detected and printed, but the function does not exit or return. (`smpc-lib/crypto/ed/commit.go#L34-L36`, `smpc-lib/crypto/ed/schnorrZK.go#L31-L33`, `smpc-lib/crypto/ed/vss.go#L46-L48`, `smpc-lib/crypto/ed/vss.go#L87-L89`)

In other similar instances, actions are either commented out or not performed:

```
nodeKey, err = crypto.GenerateKey()
if err != nil {
	//utils.Fatalf("could not generate key: %v", err)
}
if err = crypto.SaveECDSA(*genKey, nodeKey); err != nil {
	//utils.Fatalf("%v", err)
}
```

Figure 3.2: `cmd/bootnode/main.go#L61-L67`

```
case *nodeKeyHex != "":
	if nodeKey, err = crypto.HexToECDSA(*nodeKeyHex); err != nil {
		//utils.Fatalf("-nodekeyhex: %v", err)
	}
}

if *writeAddr {
	fmt.Printf("%v\n", discover.PubkeyID(&nodeKey.PublicKey))
	os.Exit(0)
}

var restrictList *netutil.Netlist
if *netrestrict != "" {
	restrictList, err = netutil.ParseNetlist(*netrestrict)
	if err != nil {
		//utils.Fatalf("-netrestrict: %v", err)
	}
}
```

```
addr, err := net.ResolveUDPAddr("udp", *listenAddr)
if err != nil {
	//utils.Fatalf("-ResolveUDPAddr: %v", err)
}
conn, err := net.ListenUDP("udp", addr)
if err != nil {
	//utils.Fatalf("-ListenUDP: %v", err)
}
```

Figure 3.3: `cmd/bootnode/main.go#L77-L103`

```
if err != nil {
	fmt.Printf("could not generate key: %v\n", err)
}
if err = crypto.SaveECDSA(genKey, nodeKey); err != nil {
	fmt.Printf("could not save key: %v\n", err)
}
```

Figure 3.4: `cmd/gsmpc/main.go#L242-L247`

```
keyjson, err = ioutil.ReadFile(*keyfile)
if err != nil {
	fmt.Println("Read keystore fail", err)
}
```

Figure 3.5: `node/config.go#L305-L329`

Exploit Scenario
An attacker is able to make the `crypto/rand` reader fail (e.g., by requesting too many bytes), and the error is unnoticed, leaving keys uninitialized and equal to zero. Causing this failure during the VSS function means that the coefficients of the random polynomial are all zero, except for the constant term, the secret.

Recommendations
Short term, in addition to logging the error, ensure that the protocol either exits the function or terminates the program with an error (depending on the case).

Long term, incorporate static analysis tools such as errcheck, Semgrep, and CodeQL in the development process to find instances in which returned errors are not checked or acted upon.

4. MtA proof verification always rejects the upper bound q^3

Severity: Informational
Difficulty: High
Type: Cryptography
Finding ID: TOB-ATSS-004
Target: `smpc-lib/crypto/ec2/{MtAZK1_nhh.go, MtAZK2_nhh.go, MtAZK3_nhh.go}`

Description
In the signing phase of the GG20 protocol, a series of multiplicative-to-additive (MtA) conversions need to be performed. These conversions use zero-knowledge range proofs, which guarantee that the value a is in the interval $[-q^3, q^3]$.

However, in the code of the three MtA conversions, the upper bound q^3 is always rejected because `S1.Cmp(N3) == 0` means that `S1` and `N3` are equal.

```
func (mtAZKProof *MtAZK1Proofnhh) MtAZK1Verifynhh(c *big.Int, publicKey *PublicKey,
ntildeH1H2 *NtildeH1H2) bool {
	if mtAZKProof.S1.Cmp(s256.S256().N3()) >= 0 { //MtAZK1 question 1
		return false
	}
```

Figure 4.1 `smpc-lib/crypto/ec2/MtAZK1_nhh.go#L89-L92`

```
func (mtAZK2Proof *MtAZK2Proofnhh) MtAZK2Verifynhh(c1 *big.Int, c2 *big.Int, publicKey
*PublicKey, ntildeH1H2 *NtildeH1H2) bool {
	if mtAZK2Proof.S1.Cmp(s256.S256().N3()) >= 0 { //MtAZK2 question 1
		return false
	}
```

Figure 4.2 `smpc-lib/crypto/ec2/MtAZK2_nhh.go#L114-L117`

```
func (mtAZK3Proof *MtAZK3Proofnhh) MtAZK3Verifynhh(c1 *big.Int, c2 *big.Int, publicKey
*PublicKey, ntildeH1H2 *NtildeH1H2) bool {
	if mtAZK3Proof.S1.Cmp(s256.S256().N3()) >= 0 { //MtAZK3 question 1
		return false
	}
```

Figure 4.3 `smpc-lib/crypto/ec2/MtAZK3_nhh.go#L122-L125`

Recommendations
Short term, replace the condition with `if proof.S1.Cmp(s256.S256().N3()) > 0`.

5. ECDSA signature verification can panic

Severity: Medium
Difficulty: Low
Type: Cryptography
Finding ID: TOB-ATSS-005
Target: `smpc-lib/smpc/common.go`

Description

The repository includes a function to verify that the ECDSA signature pair `(r, s)` for a given message and public key is valid. Because the signature verification routines compute the modular inverse of `s`, one must be cautious with the provided values. If the modular inverse of `s` does not exist, the variable `ss` becomes `nil` and a `panic()` occurs when the system attempts to multiply it with `z`.

```go
func Verify2(r *big.Int, s *big.Int, v int32, message string, pkx *big.Int, pky *big.Int)
bool {
	z, _ := new(big.Int).SetString(message, 16)
	ss := new(big.Int).ModInverse(s, secp256k1.S256().N)
	zz := new(big.Int).Mul(z, ss)
	u1 := new(big.Int).Mod(zz, secp256k1.S256().N)

	zz2 := new(big.Int).Mul(r, ss)
	u2 := new(big.Int).Mod(zz2, secp256k1.S256().N)

	if u1.Sign() == -1 {
		u1.Add(u1, secp256k1.S256().P)
	}
	ug := make([]byte, 32)
```

Figure 5.1: `smpc-lib/smpc/common.go#L120-L132`

This is caused by missing checks in the implementation; specifically, the protocol does not check that `(r, s)` is in the interval `[1, q-1]`, where `q` is the curve order, that the provided public key is on the correct elliptic curve, or that the public key is not the point at infinity.

Exploit Scenario

An attacker notices the missing checks and provides a signature pair with `s=0` for verification, causing a `panic()` on the node verifying the signature.

Recommendations

Short term, add a check to verify that `(r, s)` is in the interval `[1, q-1]`, that the provided public key is on the correct elliptic curve, and that the public key is not the point at infinity.

6. Missing checks in NtildeProof verification could lead to Golden Shoe attack

Severity: High
Type: Cryptography
Target: `smpc-lib/crypto/ec2/ntildeZK.go`

Difficulty: Low
Finding ID: TOB-ATSS-006

Description
In the signing phase of the GG20 protocol, a series of MtA conversions need to be performed. These conversion protocols involve a series of zero-knowledge proofs. As part of the setup for these proofs, each party generates an auxiliary RSA modulus, Ñ, and two random values, h_1 and h_2. When generated honestly, h_1 and h_2 generate the same subgroup modulo Ñ, the generator knows the discrete log relation between h_1 and h_2, and the auxiliary modulus is the product of two safe primes. These properties are essential for the MtA protocol, and they are enforced using the `NtildeProof`.

If these properties are not strictly enforced, it could result in a Golden Shoe attack, in which a party generates malicious Ñ, h_1, and h_2 values that allow the party to recover the secret shares of other parties.

In order to enforce these properties, each party must verify the proof and verify that h_1 and h_2 are nonzero (modulo Ñ), not equal to one (modulo Ñ), and not equal to each other (modulo Ñ). Also, they must verify that each auxiliary modulus is actually the product of two safe primes. As shown in the figures below, the proof itself is verified, but there is only a check to ensure h_1 and h_2 are not equal. Moreover, this not-equal check is not performed modulo Ñ, so it is possible to set $h_2 = h_1 + Ñ$ without being detected.

```
case *KGRound4Message:
        index := msg.GetFromIndex()
        m := msg.(*KGRound4Message)

        ///////add for ntilde zk proof check
        H1 := m.U1NtildeH1H2.H1
        H2 := m.U1NtildeH1H2.H2
        Ntilde := m.U1NtildeH1H2.Ntilde
        pf1 := m.NtildeProof1
        pf2 := m.NtildeProof2
        //fmt.Printf("=========================keygen StoreMessage, message 4, curindex = %v,
h1 = %v, h2 = %v, ntilde = %v, pf1 = %v, pf2 = %v ===========================\n", index, H1,
H2, Ntilde, pf1, pf2)
        if H1.Cmp(H2) == 0 {
                return false, errors.New("h1 and h2 were equal for this mpc node")
        }
        if !pf1.Verify(H1, H2, Ntilde) || !pf2.Verify(H2, H1, Ntilde) {
                return false, errors.New("ntilde zk proof check fail")
        }
```

Figure 6.1: `smpc-lib/ecdsa/keygen/local_dnode.go#L211-L227`

```
// Verify Verify ntilde proof
```

```
func (p *NtildeProof) Verify(h1, h2, N *big.Int) bool {
	if p == nil {
		return false
	}
	modN := ModInt(N)
	msg := append([]*big.Int{h1, h2, N}, p.Alpha[:]...)
	c := Sha512_256i(msg...)
	cIBI := new(big.Int)
	for i := 0; i < Iterations; i++ {
		if p.Alpha[i] == nil || p.T[i] == nil {
			return false
		}
		cI := c.Bit(i)
		cIBI = cIBI.SetInt64(int64(cI))
		h1ExpTi := modN.Exp(h1, p.T[i])
		h2ExpCi := modN.Exp(h2, cIBI)
		alphaIMulH2ExpCi := modN.Mul(p.Alpha[i], h2ExpCi)
		if h1ExpTi.Cmp(alphaIMulH2ExpCi) != 0 {
			return false
		}
	}
	return true
}
```

Figure 6.2: `smpc-lib/crypto/ec2/ntildeZK.go#L64-L87`

Exploit Scenario

An attacker notices that h_1 and h_2 can be zero, one, or equal to each other modulo Ñ, and that there is no check to ensure Ñ is the product of two safe primes. He generates malicious Ñ, h_1, and h_2 values and performs a Golden Shoe attack against the other parties.

Recommendations

Short term, change the comparison between h_1 and h_2 to be performed modulo Ñ. Also, add checks to ensure that h_1 and h_2 are not equal to zero or one modulo Ñ. Finally, have the prover include a proof that Ñ is the product of two primes using the proof system from Section 3.4 of Goldberg et al. (2019). Ensure that the verifier verifies this proof as well.

Long term, review all zero-knowledge proof verifications to ensure they properly validate all input values.

7. Insufficient input validation in Feldman VSS operations could result in trivial or unrecoverable shares

Severity: Low
Difficulty: Low
Type: Cryptography
Finding ID: TOB-ATSS-007
Target: `smpc-lib/crypto/ed/vss.go`

Description

The codebase uses the Feldman VSS scheme in various locations to distribute secret shares of secret data used in the threshold signing protocol. The `Vss` and `Vss2` functions take as input the secret data, the size of the group (`n`), the threshold for the group (`t`), and the group IDs.

The threshold represents how many parties need to combine their shares to recover the secret. If this threshold is less than two, then this function will return the secret data itself. If this threshold is larger than the size of the group, then it will be impossible to recover the secret data. However, the `Vss` and `Vss2` do not validate the threshold value, so these functions could return either trivial or unrecoverable shares.

```
// Vss  Calculate secret sharing value
func Vss(secret [32]byte, ids [][32]byte, t int, n int) ([][32]byte, [][32]byte, [][32]byte)
{

        var cfs, cfsBBytes, shares [][32]byte

        cfs = append(cfs, secret)

        var cfB ExtendedGroupElement
        var cfBBytes [32]byte
        GeScalarMultBase(&cfB, &secret)
        cfB.ToBytes(&cfBBytes)
        cfsBBytes = append(cfsBBytes, cfBBytes)

        var zero [32]byte
        var one [32]byte
        one[0] = 1
        rand := cryptorand.Reader

        for i := 1; i <= t-1; i++ {
                var rndNum [32]byte
                if _, err := io.ReadFull(rand, rndNum[:]); err != nil {
                        fmt.Println("Error: io.ReadFull(rand, rndNum[:])")
                }
                ScMulAdd(&rndNum, &rndNum, &one, &zero)

                cfs = append(cfs, rndNum)

                GeScalarMultBase(&cfB, &rndNum)
                cfB.ToBytes(&cfBBytes)
                cfsBBytes = append(cfsBBytes, cfBBytes)
        }

        for i := 0; i < n; i++ {
```

```
		share := calculatePolynomial(cfs, ids[i])
		shares = append(shares, share)
	}

	return cfs, cfsBBytes, shares
}
```

Figure 7.1: `smpc-lib/crypto/ed/vss.go#L26-L64`

```
// Vss2  Calculate secret sharing value
func Vss2(secret [32]byte, t int, n int, uids map[string][32]byte) ([][32]byte, [][32]byte,
map[string][32]byte) {

	var cfs, cfsBBytes [][32]byte
	var shares = make(map[string][32]byte)

	cfs = append(cfs, secret)

	var cfB ExtendedGroupElement
	var cfBBytes [32]byte
	GeScalarMultBase(&cfB, &secret)
	cfB.ToBytes(&cfBBytes)
	cfsBBytes = append(cfsBBytes, cfBBytes)

	var zero [32]byte
	var one [32]byte
	one[0] = 1
	rand := cryptorand.Reader

	for i := 1; i <= t-1; i++ {
		var rndNum [32]byte
		if _, err := io.ReadFull(rand, rndNum[:]); err != nil {
			fmt.Println("Error: io.ReadFull(rand, rndNum[:])")
		}
		ScMulAdd(&rndNum, &rndNum, &one, &zero)

		cfs = append(cfs, rndNum)

		GeScalarMultBase(&cfB, &rndNum)
		cfB.ToBytes(&cfBBytes)
		cfsBBytes = append(cfsBBytes, cfBBytes)
	}

	for k, v := range uids {
		share := calculatePolynomial(cfs, v)
		shares[k] = share
	}

	return cfs, cfsBBytes, shares
}
```

Figure 7.2: `smpc-lib/crypto/ed/vss.go#L66-L105`

In addition to the `Vss` and `Vss2` functions' lack of input validation, there is insufficient validation of these values at the `p2p` layer. For instance, the current check allows the threshold to be a negative number, which would result in trivial secret shares.

```
func CheckAddPeer(threshold string, enodes []string, subGroup bool) (bool, error) {
        thshall := false
        es := strings.Split(threshold, "/")
        if len(es) != 2 {
                msg := fmt.Sprintf("args threshold(%v) format is wrong", threshold)
                return thshall, errors.New(msg)
        }
        nodeNum0, _ := strconv.Atoi(es[0])
        nodeNum1, _ := strconv.Atoi(es[1])
        if len(enodes) < nodeNum0 || len(enodes) > nodeNum1 {
                msg := fmt.Sprintf("args threshold(%v) and enodes(%v) not match", threshold,
enodes)
                return thshall, errors.New(msg)
        }
```

Figure 7.3: `smpc-lib/p2p/layer2/smpc.go#L387-L399`

Exploit Scenario
An attacker notices the missing validation in `Vss` and `Vss2` and causes the parties to create secret shares using a threshold value that is larger than the group size. As a result, each party generates unrecoverable shares, and the protocol aborts.

Recommendations
Short term, add checks in `Vss`, `Vss2`, and `CheckAddPeer` to ensure that `t` is greater than one and less than or equal to `n`.

Long term, review all functions for critical components of the protocol and ensure they have comprehensive input validation.

8. Missing domain separation in various hash computations could result in canonicalization attacks

Severity: High
Difficulty: High
Type: Cryptography
Finding ID: TOB-ATSS-008
Target: `crypto/ec2/commit.go`, various

Description

The codebase uses cryptographic hash functions (mostly SHA-3) in various locations. For example, SHA-3 is used as part of the commitment scheme, and it is used for generating Fiat-Shamir challenges in the MtA and other proof systems. When calculating a hash value from multiple inputs, it is best practice to use a domain separator, a custom string to separate inputs, especially when the inputs do not have fixed sizes. For example, when computing the hash of inputs `A` and `B`, the system should compute `SHA-3(A || “my_custom_domain_separator” || B)`, rather than `SHA-3(A || B)`.

Using a domain separator will help protect against canonicalization attacks, in which an attacker finds a hash collision between two different inputs by moving a substring at the end of the first input to the beginning of the second input.

This missing domain separation could be problematic in a few locations. For example, as shown in figure 8.1, the commitment scheme uses SHA-3 without a domain separator to form a commitment from an array of secret values. Moreover, the hash is computed by concatenating the byte representation of the secret values, which are of type `big.Int`. When `big.Int` values are converted into bytes, the length of the byte string will depend on the size of the `big.Int` value. In other words, these secret values will not have a fixed size, so a canonicalization attack is possible.

```
// Commit  Generate commitment data by secrets
func (commitment *Commitment) Commit(secrets ...*big.Int) *Commitment {
	// Generate the random num
	rnd := random.GetRandomInt(256)
	if rnd == nil {
		return nil
	}

	// First, hash with the keccak256
	sha3256 := sha3.New256()
	//keccak256 := sha3.NewKeccak256()

	sha3256.Write(rnd.Bytes())

	for _, secret := range secrets {
		sha3256.Write(secret.Bytes())
	}

	digestKeccak256 := sha3256.Sum(nil)

	//second, hash with the SHA3-256
	sha3256.Write(digestKeccak256)
```

```
        digest := sha3256.Sum(nil)

        // convert the hash ([]byte) to big.Int
        digestBigInt := new(big.Int).SetBytes(digest)

        D := []*big.Int{rnd}
        D = append(D, secrets...)

        commitment.C = digestBigInt
        commitment.D = D

        return commitment
}
```
Figure 8.1: `smpc-lib/crypto/ec2/commit.go#L33-L67`

The same issue applies to all of the Fiat-Shamir challenges computed in various proof systems, as well as other instances in which SHA-3 is used.

Exploit Scenario
An attacker notices that various hash computations do not use domain separators. She is then able to perform a canonicalization attack on a commitment scheme and decommit to a different value.

Recommendations
Short term, pick a custom domain separator specific to AnySwap and this protocol. Identify all instances in which a hash is computed over multiple inputs that do not have a fixed size, and add this domain separator to these instances.

Long term, add the custom domain separator to all hash computations.

9. Weak Fiat-Shamir transformation in MtA proofs

Severity: High
Difficulty: High
Type: Cryptography
Finding ID: TOB-ATSS-009
Target: `smpc-lib/crypto/ec2/{MtAZK1_nhh.go`, `MtAZK2_nhh.go`, `MtAZK3_nhh.go}`

Description

The `smpc-lib/crypto/` folder contains implementations of several zero-knowledge proof systems required in the threshold signature protocol GG20. These zero-knowledge proof systems are essential to the security guarantees of the protocol, ensuring that the parties remain honest. There are three zero-knowledge range-proofs required to implement the MtA share conversion.

All implementations follow the Fiat-Shamir transformation and use a hash function to non-interactively generate the verifier challenge `e`. However, in all three proofs, the hash calculation does not include the entire public statement of the proof.

```
sha3256 := sha3.New256()
sha3256.Write(z.Bytes())
sha3256.Write(u.Bytes())
sha3256.Write(w.Bytes())

sha3256.Write(publicKey.N.Bytes()) //MtAZK1 question 2

eBytes := sha3256.Sum(nil)
```

Figure 9.1: `smpc-lib/crypto/ec2/MtAZK1_nhh.go`

In `MtAZK1_nhh.go`, the range proof proves that the plaintext of a given ciphertext, `c`, lies within some predefined range. To prevent forgeries, it is important that the hash function used for generating verifier challenge values includes the `c` variable corresponding to the ciphertext.

```
sha3256 := sha3.New256()
sha3256.Write(z.Bytes())
sha3256.Write(zBar.Bytes())
sha3256.Write(t.Bytes())
sha3256.Write(v.Bytes())
sha3256.Write(w.Bytes())

sha3256.Write(publicKey.N.Bytes()) //MtAZK2 question 2

eBytes := sha3256.Sum(nil)
```

Figure 9.2: `smpc-lib/crypto/ec2/MtAZK2_nhh.go`

Likewise, in `MtAZK2_nhh.go`, the hash function should also include the c_1 and c_2 variables related to the public statement of the proof.

```
sha3256 := sha3.New256()
sha3256.Write(ux.Bytes())
```

```
sha3256.Write(uy.Bytes())
sha3256.Write(z.Bytes())
sha3256.Write(zBar.Bytes())
sha3256.Write(t.Bytes())
sha3256.Write(v.Bytes())
sha3256.Write(w.Bytes())

sha3256.Write(publicKey.N.Bytes()) //MtAZK3 question 2

eBytes := sha3256.Sum(nil)
```

Figure 9.3: `smpc-lib/crypto/ec2/MtAZK3_nhh.go`

In `MtAZK3_nhh.go`, in addition to the c_1 and c_2 variables missing from the `MtAZK2_nhh.go` proof, the hash function should include the public discrete logarithm present in the proof statement.

Exploit Scenario
An attacker notices that these proof systems do not include the parts of the public statement in the Fiat-Shamir challenge and can forge proofs, undermining the security guarantees of the protocol.

Recommendations
Short term, include the public statement values in the computation of the hash function.

Long term, review all utilizations of the Fiat-Shamir transformation for missing public parameters and use a domain separator to separate each value.

10. Nonbinding "MtA with check" proof

Severity: Medium
Type: Cryptography
Target: `smpc-lib/crypto/ec2/MtAZK3_nhh.go`

Difficulty: Low
Finding ID: TOB-ATSS-010

Description
In the signing phase of the GG20 protocol, a series of MtA conversions need to be performed. The third MtA conversion described in the specification is augmented with a check, which, along with including a range proof, provides proof of knowledge of a discrete logarithm. This proof of knowledge guarantees that the prover used his w_j variable as input to the conversion protocol.

However, the implementation of `MtAZK3_nhh.go` does not include such a discrete logarithm proof of knowledge and differs from the implementation in `MtAZK2_nhh.go` only in the inclusion of two big integers equal to zero on the computation of the hash function.

```
// ux, uy := s256.S256().ScalarBaseMult(alpha.Bytes())
ux := big.NewInt(0)
uy := big.NewInt(0)
```

Figure 10.1: `smpc-lib/crypto/ec2/MtAZK3_nhh.go#L58-L60`

Exploit Scenario
An attacker notices that this proof is not binding to his public discrete logarithm value and uses this fact to create an "MtA with check" proof for a value different from his private share. As a result, the attacker prolongs the protocol for more rounds before the other participants abort and gains information about the others' secrets. The attacker also avoids being detected as the party causing the abort, since the "MtA with check" proof is one of the properties required for an identifiable abort.

Recommendations
Short term, include the discrete logarithm statement in the "MtA with check" proof according to the proof described in Appendix A.2 of the GG18 paper. Notice the typo in the paper's verification description: where it reads "The Verifier checks that $s_1 \le q^3$, $g^1 = X^e u \in G$," it should read "The Verifier checks that $s_1 \le q^3$, $g^{s1} = X^e u \in G$."

11. Missing implementation of phase 5 of the signing protocol

Severity: Medium
Difficulty: High
Type: Cryptography
Finding ID: TOB-ATSS-011
Target: `smpc-lib/ecdsa/signing/round_7.go`

Description
In the signing protocol of the GG20 spec, several phases ensure that participants behave honestly. Phase 5 of the protocol requires participants to broadcast a value and a consistency proof between this value and a previously sent value for MtA.

However, after computing `R = Gamma`$^{\texttt{inv(delta)}}$, the implementation in `round_7.go` is missing the last phases of the specification, for both the “Simplified one round online ECDSA” and the “One-Round Threshold ECDSA with identifiable abort” versions of the protocol. Participants simply compute their $\texttt{s}_{\texttt{i}}$ values in the implementation and do not verify whether they correctly calculated the `R` value.

Exploit Scenario
An attacker notices that phases 5 and 6 are not included in the implementation. Since there is not a proof of consistency with the messages sent during the MtA protocol, the attacker is able to send malicious shares and violate the security of the protocol.

Recommendations
Short term, implement phases 5 and 6 as described in section 5.1 of the GG20 spec.

12. Paillier modulus is not proved to be square-free

Severity: Medium
Type: Cryptography
Target: `smpc-lib/crypto/ec2/paillier.go`

Difficulty: Low
Finding ID: TOB-ATSS-012

Description
Participants generate a Paillier key E_i to be used during the signing protocol in the key generation and reshare protocols. As stated in section 3.1 of the GG20 paper, in phase 3 of the key generation protocol, participants need to show that E_i is a square-free integer. This proof is missing from the implementation.

Additionally, the `GenerateKeyPair` function does not check that `p` and `q` are equal. As a result, honest participants could generate a perfect square as their key.

```
// GenerateKeyPair create paillier pubkey and private key
func GenerateKeyPair(length int) (*PublicKey, *PrivateKey) {
	one := big.NewInt(1)

	sp1 := <-SafePrimeCh
	p := sp1.p
	sp2 := <-SafePrimeCh
	q := sp2.p

	if p == nil || q == nil {
		return nil, nil
	}

	SafePrimeCh <- sp1
	SafePrimeCh <- sp2

	n := new(big.Int).Mul(p, q)
```

Figure 12.1: `smpc-lib/crypto/ec2/paillier.go#L49-L65`

Exploit Scenario
An attacker notices that there is no proof showing that each party's Paillier key is square-free. She generates a malicious key that is not square-free, and the security of the protocol (which relies on the assumption that these are square-free) is violated.

Recommendations
Short term, add a zero-knowledge proof that E_i is a square-free integer, such as the proof in section 3.1 of Gennaro et al. (1998), or the proof in section 3.2 of Goldberg et al. (2019). Additionally, add a check to the `GenerateKeyPair` function to verify that the generated prime numbers are not equal.

13. Unnecessary key generation in the reshare protocol

Severity: Informational
Difficulty: Low
Type: Cryptography
Finding ID: TOB-ATSS-013
Target: `smpc-lib/ecdsa/reshare/{round_3.go, round_4.go}`

Description

The reshare protocol allows a party to include or remove members from the current group. To do this, users generate new VSS polynomials to share the secret, considering the new number of members. Afterward, the parties share the VSS commitments and the shared secret, and the new party members generate new Paillier keys and Fujisaki-Okamoto commitments Ñ, h_1, and h_2.

The Paillier keys and the Ñ, h_1, and h_2 values take a long time to generate and are not required to be generated by old party members, but in the implementation, all members renew their values.

```
u1PaillierPk, u1PaillierSk := ec2.GenerateKeyPair(round.paillierkeylength)

//round.Save.U1PaillierSk = u1PaillierSk
//round.Save.U1PaillierPk[curIndex] = u1PaillierPk
round.temp.u1PaillierSk = u1PaillierSk
round.temp.u1PaillierPk = u1PaillierPk

re := &ReRound3Message{
	ReRoundMessage: new(ReRoundMessage),
	U1PaillierPk:       u1PaillierPk,
}
re.SetFromID(round.dnodeid)
re.SetFromIndex(curIndex)
round.temp.reshareRound3Messages[curIndex] = re
round.out <- re
```

Figure 13.1: Generation of the Paillier keys
(smpc-lib/ecdsa/reshare/round_3.go#L132-L146)

```
NtildeLength := 2048
u1NtildeH1H2, alpha, beta, p, q := ec2.GenerateNtildeH1H2(NtildeLength)
if u1NtildeH1H2 == nil {
	return errors.New("gen ntilde h1 h2 fail")
}

ntildeProof1 := ec2.NewNtildeProof(u1NtildeH1H2.H1, u1NtildeH1H2.H2, alpha, p, q,
u1NtildeH1H2.Ntilde)
ntildeProof2 := ec2.NewNtildeProof(u1NtildeH1H2.H2, u1NtildeH1H2.H1, beta, p, q,
u1NtildeH1H2.Ntilde)

re := &ReRound4Message{
	ReRoundMessage: new(ReRoundMessage),
	U1NtildeH1H2:       u1NtildeH1H2,
	NtildeProof1:       ntildeProof1,
	NtildeProof2:       ntildeProof2,
}
```

```
re.SetFromID(round.dnodeid)
re.SetFromIndex(curIndex)

round.temp.u1NtildeH1H2 = u1NtildeH1H2
round.temp.reshareRound4Messages[curIndex] = re
round.out <- re
```

Figure 13.1: Generation of the Paillier keys
(`smpc-lib/ecdsa/reshare/round_4.go#L53-L73`)

Recommendations
Short term, revise the implementation so that the Paillier keys and Fujisaki-Okamoto commitments Ñ, h_1, and h_2 have to be generated only for new party members.

14. Fiat-Shamir challenge in MtA proofs use the wrong modulus

Severity: Informational
Difficulty: N/A
Type: Cryptography
Finding ID: TOB-ATSS-014
Target: `smpc-lib/crypto/ec2/{MtAZK1_nhh.go`, `MtAZK2_nhh.go`, `MtAZK3_nhh.go}`

Description

Each of the three MtA proof systems use the Fiat-Shamir transformation to non-interactively generate verifier challenge value e. For each of the proof systems, the protocol specifies that these values should be random values smaller than q, the order of the DSA group.

```
// MtAZK1Provenhh  Generate zero knowledge proof data mtazk1proof_ nhh
func MtAZK1Provenhh(m *big.Int, r *big.Int, publicKey *PublicKey, ntildeH1H2 *NtildeH1H2)
*MtAZK1Proofnhh {

	...

	sha3256 := sha3.New256()
	sha3256.Write(z.Bytes())
	sha3256.Write(u.Bytes())
	sha3256.Write(w.Bytes())

	sha3256.Write(publicKey.N.Bytes()) //MtAZK1 question 2

	eBytes := sha3256.Sum(nil)
	e := new(big.Int).SetBytes(eBytes)

	e = new(big.Int).Mod(e, publicKey.N)
```

Figure 14.1: `smpc-lib/crypto/ec2/MtAZK1_nhh.go#L40-L72`

As shown in figure 14.1, the implementation currently generates a value smaller than the `publicKey.N` value, not q, the order of the DSA group. Instead, this line should reduce e modulo `s256.S256().N`. This applies to all three proof systems, but it does not appear to be immediately exploitable.

Recommendations

Short term, adjust the implementation of all three MtA proof systems to reduce e modulo `s256.S256().N`.

15. MtA proof verification implicitly trusts the prover's Paillier public key

Severity: High
Difficulty: Low
Type: Cryptography
Finding ID: TOB-ATSS-015
Target: `smpc-lib/crypto/ec2/{MtAZK1_nhh.go`, `MtAZK2_nhh.go`, `MtAZK3_nhh.go}`

Description

For each of the three MtA proof systems, the prover submits a series of values along with his Paillier public key. The Paillier public key contains three values: the modulus (`N`), the modulus squared (`N2`), and the public parameter (`G`). When Paillier public keys are generated in this implementation, `G` is set to be equal to `N + 1`, which is common for most implementations.

Therefore, the Paillier public key consists of a modulus, `N`, and two other values derived from `N`. In the verification functions for each proof system, the verifier performs a series of checks using the submitted values along with this public key. In the current implementation, the verifier simply uses the `N2` and `G` values submitted by the prover, implicitly trusting that these values are valid.

```
// MtAZK1Verifynhh  Verify zero knowledge proof data mtazk1proof_ nhh
func (mtAZKProof *MtAZK1Proofnhh) MtAZK1Verifynhh(c *big.Int, publicKey *PublicKey,
ntildeH1H2 *NtildeH1H2) bool {
	if mtAZKProof.S1.Cmp(s256.S256().N3()) >= 0 { //MtAZK1 question 1
		return false
	}

	sha3256 := sha3.New256()
	sha3256.Write(mtAZKProof.Z.Bytes())
	sha3256.Write(mtAZKProof.U.Bytes())
	sha3256.Write(mtAZKProof.W.Bytes())

	sha3256.Write(publicKey.N.Bytes()) //MtAZK1 question 2

	eBytes := sha3256.Sum(nil)
	e := new(big.Int).SetBytes(eBytes)

	e = new(big.Int).Mod(e, publicKey.N)

	u2 := new(big.Int).Exp(publicKey.G, mtAZKProof.S1, publicKey.N2)
	u2 = new(big.Int).Mul(u2, new(big.Int).Exp(mtAZKProof.S, publicKey.N, publicKey.N2))
	u2 = new(big.Int).Mod(u2, publicKey.N2)
	// *****
	ce := new(big.Int).Exp(c, e, publicKey.N2)
	ceU := new(big.Int).Mul(ce, mtAZKProof.U)
	ceU = new(big.Int).Mod(ceU, publicKey.N2)
```

Figure 15.1: `smpc-lib/crypto/ec2/MtAZK1_nhh.go#L40-L72`

In general, it is considered best practice to reconstruct values rather than parsing and validating them. Therefore, rather than simply using these `N2` and `G` values, it would be safer to reconstruct them manually from the `N` value submitted by the prover. This will prevent attacks in which the prover uses unexpected `N2` or `G` values.

Exploit Scenario
An attacker realizes that the MtA proof verification implicitly trusts the Paillier public key submitted with the proof. He submits a proof using incorrect `N2` and `G` values to forge the MtA proofs.

Recommendations
Short term, adjust the implementation of all three MtA proof verifiers to reconstruct the `N2` and `G` values rather than using the values submitted by the prover.

16. Insufficient input validation in MtA verify functions could allow proof forgery

Severity: Low
Difficulty: Low
Type: Cryptography
Finding ID: TOB-ATSS-016
Target: `smpc-lib/crypto/ec2/{MtAZK1_nhh.go`, `MtAZK2_nhh.go`, `MtAZK3_nhh.go}`

Description

As part of the signing process, a series of MtA conversions are performed. These conversions involve three different zero-knowledge proofs to ensure that each party is behaving honestly. In order to ensure this honesty, it is imperative that each of these proofs are verified carefully with strong restrictions on the proof input values. However, the current implementations of each of these proofs have insufficient input validation, which could result in proof forgeries.

```
func (mtAZKProof *MtAZK1Proofnhh) MtAZK1Verifynhh(c *big.Int, publicKey *PublicKey,
ntildeH1H2 *NtildeH1H2) bool {
       if mtAZKProof.S1.Cmp(s256.S256().N3()) >= 0 { //MtAZK1 question 1
              return false
       }

       sha3256 := sha3.New256()
       sha3256.Write(mtAZKProof.Z.Bytes())
       sha3256.Write(mtAZKProof.U.Bytes())
       sha3256.Write(mtAZKProof.W.Bytes())

       sha3256.Write(publicKey.N.Bytes()) //MtAZK1 question 2

       eBytes := sha3256.Sum(nil)
       e := new(big.Int).SetBytes(eBytes)

       e = new(big.Int).Mod(e, publicKey.N)

       u2 := new(big.Int).Exp(publicKey.G, mtAZKProof.S1, publicKey.N2)
       u2 = new(big.Int).Mul(u2, new(big.Int).Exp(mtAZKProof.S, publicKey.N, publicKey.N2))
       u2 = new(big.Int).Mod(u2, publicKey.N2)
       // *****
       ce := new(big.Int).Exp(c, e, publicKey.N2)
       ceU := new(big.Int).Mul(ce, mtAZKProof.U)
       ceU = new(big.Int).Mod(ceU, publicKey.N2)

       if ceU.Cmp(u2) != 0 {
              return false
       }

       w2 := new(big.Int).Exp(ntildeH1H2.H1, mtAZKProof.S1, ntildeH1H2.Ntilde)
       w2 = new(big.Int).Mul(w2, new(big.Int).Exp(ntildeH1H2.H2, mtAZKProof.S2,
ntildeH1H2.Ntilde))
       w2 = new(big.Int).Mod(w2, ntildeH1H2.Ntilde)
       // *****
       ze := new(big.Int).Exp(mtAZKProof.Z, e, ntildeH1H2.Ntilde)
       zeW := new(big.Int).Mul(mtAZKProof.W, ze)
       zeW = new(big.Int).Mod(zeW, ntildeH1H2.Ntilde)
```

```
	if zeW.Cmp(w2) != 0 {
		return false
	}

	return true
}
```

Figure 16.1: `smpc-lib/crypto/ec2/MtAZK1_nhh.go#L88-L131`

The verification function for the first proof system is shown in figure 16.1. This function has insufficient input validation. First, there are no restrictions on the `Z`, `U`, and `W` input values. The function should ensure that each of these values is in the expected range (`Z` and `W` should be less than `Ntilde`, and `U` should be less than `publicKey.N2`). Since the verification is performed using modular arithmetic, larger `Z`, `U`, and `W` values may not affect the arithmetic of the verification; however, the lack of validation on these values could allow the prover use `Z`, `U`, or `W` values that are equivalent modulo `Ntilde` (or `publicKey.N2` for `U`) to generate multiple different `e` challenge values. With the option of generating different `e` values, a malicious prover could search for an `e` value that allows for proof forgery (though this would be difficult in practice). Lastly, the verification function also needs to check that `S` is less than `publicKey.N`.

In addition to these missing checks, checks ensuring that all of the input values are nonzero and not equal to one are missing (these checks need to be performed modularly). Currently, a malicious prover could set the `Z` and `W` input values equal to one and the `U`, `S`, `S1`, and `S2` values equal to zero, and the proof will pass verification.

Similar issues also affect the second and third proof systems. For brevity, we omit the code for the verification functions, but the proof systems are similar and need to perform the same checks:

- Check that `Z`, `ZBar`, `T`, and `W` are less than `Ntilde`
- Check that `V`, `c1`, and `c2` are less than `publicKey.N2`
- Check that `S` is less than `publicKey.N`
- Check that `c1`, `c2`, `Z`, `ZBar`, `T`, `V`, `W`, `S`, `S1`, `S2`, `T1`, and `T2` are not equal (modularly) to zero or one

Exploit Scenario
An attacker notices that these verification functions do not validate their input values. She submits proofs in which all of the values are either zero or one, which pass verification. This allows her to perform the MtA conversion maliciously.

Recommendations
Short term, add the following checks to the MtA verification functions for the first proof system:

- Check that `Z` and `W` are less than `Ntilde`
- Check that `U` and `c` are less than `publicKey.N2`
- Check that `S` is less than `publicKey.N`
- Check that `c`, `Z`, `U`, `W`, `S`, `S1`, and `S2` are not modularly equal to zero or one

Add the following checks for the second and third proof systems:

- Check that `Z`, `ZBar`, `T`, and `W` are less than `Ntilde`
- Check that `V`, `c1`, and `c2` are less than `publicKey.N2`
- Check that `S` is less than `publicKey.N`
- Check that `c1`, `c2`, `Z`, `ZBar`, `T`, `V`, `W`, `S`, `S1`, `S2`, `T1`, and `T2` are not modularly equal to zero or one

Long term, review all functions for critical components of the protocol and ensure they have comprehensive input validation.

17. Key generation includes the wrong Schnorr proof

Severity: Informational	Difficulty: N/A
Type: Cryptography	Finding ID: TOB-ATSS-017
Target: `smpc-lib/crypto/ecdsa/keygen/round_5.go`	

Description
The threshold signature scheme includes a key generation protocol in which each party generates a share of the group's signing key. To ensure that each party is behaving honestly, this protocol includes a combination of commitments and zero-knowledge proofs. However, the implementation currently includes a different zero-knowledge proof than specified.

In phase 1 of the key generation protocol, each party randomly generates secret values, u_i. Later in this protocol, these and other values are used to generate secret values, x_i, which are the final secret shares of the group signing key. The specification then calls for each party to produce a Schnorr proof of knowledge on these x_i values in phase 3. However, the implementation currently includes a Schnorr proof of the u_i values instead.

```
// Start broacast zku proof data
func (round *round5) Start() error {
	if round.started {
		return errors.New("round already started")
	}
	round.number = 5
	round.started = true
	round.resetOK()

	curIndex, err := round.GetDNodeIDIndex(round.dnodeid)
	if err != nil {
		return err
	}

	u1zkUProof := ec2.ZkUProve(round.temp.u1)
```
Figure 17.1: `smpc-lib/crypto/ecdsa/keygen/round_5.go#L25-L39`

Exploit Scenario
An attacker notices that the key generation protocol includes the wrong Schnorr proof of knowledge. Since this is a significant deviation from the specification, the current security proof does not apply, and the attacker is able to violate the security of the protocol.

Recommendations
Short term, replace the Schnorr proof of knowledge of u_i with a proof of knowledge of x_i.

18. Incoming messages are not validated and could cause crashes

Severity: Medium
Difficulty: Low
Type: Data Validation
Finding ID: TOB-ATSS-018
Target: `smpc/{key_ec.go, key_ed.go, sign_ed.go, sign_ec.go, reshare.go}`

Description

During the execution of the key generation, signing, and resharing protocols, participants send broadcast and peer-to-peer messages. When the implementation handles messages from the network, it needs to convert the fields from strings to their correct types. The implementation assumes that the messages contain fields and that big integers are initialized successfully from the data in the messages.

If a malicious participant sends a message with a field that is not parseable to an integer in base 10 (or omits the field in the outbound map), the variable will contain `nil`, which can cause crashes and generate incorrect values during the protocol.

For example, if `ComC` were omitted from the outbound message, the incoming parsed `ComC` value would be `nil`, causing a null dereference in round 4 of the key generation process when the commit is checked.

```
//1 message
if msg["Type"] == "KGRound1Message" {
	pub := &ec2.PublicKey{}
	err := pub.UnmarshalJSON([]byte(msg["U1PaillierPk"]))
	if err == nil {
		comc, _ := new(big.Int).SetString(msg["ComC"], 10)
		ComCBip32, _ := new(big.Int).SetString(msg["ComC_bip32"], 10)
		kg := &keygen.KGRound1Message{
			KGRoundMessage: new(keygen.KGRoundMessage),
			ComC:           comc,
			ComCBip32:      ComCBip32,
			U1PaillierPk:   pub,
		}
		kg.SetFromID(from)
		kg.SetFromIndex(index)
		kg.ToID = to
		return kg
	}
}
```

Figure 18.1: `smpc/key_ec.go#L101-L119`

This value will be used directly in round 4, which does not contain additional checks, leading to a crash.

```
deCommit := &ec2.Commitment{C: msg1.ComC, D: msg3.ComU1GD}
_, u1G := deCommit.DeCommit()
```

Figure 18.2: `smpc-lib/ecdsa/keygen/round_4.go#L111-L112`

```
panic: runtime error: invalid memory address or nil pointer dereference
[signal SIGSEGV: segmentation violation code=0x1 addr=0x0 pc=0x515938]

goroutine 4437 [running]:
math/big.(*Int).Cmp(0x4003391a78, 0x0)
	math/big/int.go:328 +0x38
github.com/anyswap/Anyswap-MPCNode/smpc-lib/crypto/ec2.(*Commitment).Verify(0x4003391cc8)
	github.com/anyswap/Anyswap-MPCNode/smpc-lib/crypto/ec2/commit.go:84 +0x1ec
github.com/anyswap/Anyswap-MPCNode/smpc-lib/ecdsa/keygen.(*round4).Start(0x40033c61c8)
	github.com/anyswap/Anyswap-MPCNode/smpc-lib/ecdsa/keygen/round_4.go:72 +0x348
github.com/anyswap/Anyswap-MPCNode/smpc-lib/smpc.BaseUpdate({0x15204c0, 0x40042de1c0},
{0x1509c18, 0x4003ed34a0})
	github.com/anyswap/Anyswap-MPCNode/smpc-lib/smpc/dnode.go:127 +0x220
github.com/anyswap/Anyswap-MPCNode/smpc-lib/ecdsa/keygen.(*LocalDNode).Update(0x40042de1c0,
{0x1509c18, 0x4003ed34a0})
	github.com/anyswap/Anyswap-MPCNode/smpc-lib/ecdsa/keygen/local_dnode.go:136 +0x44
github.com/anyswap/Anyswap-MPCNode/smpc.ProcessInboundMessages({0x4002f3a0f0, 0x42},
0x40005b5bc0, 0x400422dea0, 0x4002c45560)
	github.com/anyswap/Anyswap-MPCNode/smpc/key_ec.go:75 +0x9a0
created by github.com/anyswap/Anyswap-MPCNode/smpc.KeyGenerateDECDSA
	github.com/anyswap/Anyswap-MPCNode/smpc/reqaddr.go:732 +0x5c0
```

Figure 18.2: Crash trace when an empty `ComC` value is sent

Exploit Scenario
An attacker sends a malformed message to another participant, causing the participant to crash. The attacker uses the vulnerabilities described in TOB-ATSS-019 and TOB-ATSS-020 to hijack that participant's session and pretend to be him for the remainder of the protocol.

Recommendations
Short term, implement a mechanism to check that all incoming messages have the required fields and that all values are correctly parsed in `smpc/{key_ec.go`, `key_ed.go`, `sign_ed.go`, `sign_ec.go`, `reshare.go}`.

19. In the transport layer, broadcast messages are not signed

Severity: High
Difficulty: Low
Type: Cryptography
Finding ID: TOB-ATSS-019
Target: Several files

Description
The repository implements a multiparty threshold signature scheme, which includes the key generation, signing, and resharing protocols. In a multiparty scheme, participants exchange messages in peer-to-peer channels and broadcast channels. These channels have to satisfy security properties to prevent attackers from interfering with the normal execution of the protocol.

In particular, broadcast channels must ensure that all participants receive the same messages and that a sender signs his message. When handling incoming messages from the broadcast channel, participants must verify the message signature and check that it comes from the user in the `fromId` field in the message.

The details of the broadcast channels are not documented. We asked AnySwap for more information, and the team informed us that broadcast messages are not signed; this means that an adversary could alter or forge other parties' messages.

Exploit Scenario
A malicious player broadcasts messages as another participant by sending a message with a different `fromId` field than his own.

Recommendations
Short term, add a check to ensure that participants sign their broadcasted messages. Additionally, ensure that when the system handles incoming messages, it ensures the validity of the signature and checks that the signature comes from the user in the `fromId` field in the message.

Long term, document the details of both the broadcast and peer-to-peer channels.

20. In the transport layer, peer-to-peer messages are not signed

Severity: High
Type: Cryptography
Target: Several files

Difficulty: Low
Finding ID: TOB-ATSS-020

Description
The repository implements a multiparty threshold signature scheme, which includes the key generation, signing, and resharing protocols. In a multiparty scheme, participants exchange messages in peer-to-peer channels and broadcast channels. These channels have to satisfy security properties to prevent attackers from interfering with the normal execution of the protocol.

In particular, messages exchanged in the peer-to-peer channels must be signed by the sender and encrypted with the receiver's key. When the system handles incoming messages from another peer, besides checking that the message correctly decrypts, it must check that the signature is valid and that it matches the node's public key in the `fromId` field.

The details of the peer-to-peer channels are not documented. We asked AnySwap for more information, and the team informed us that peer-to-peer messages are encrypted but not signed; this means that an adversary could alter or forge other parties' messages.

Exploit Scenario
A malicious player sends another player messages as another participant by sending a message with a different `fromId` field than his own.

Recommendations
Short term, add a check to ensure that participants sign their peer-to-peer messages before encrypting them. Additionally, ensure that when the system handles incoming messages, it decrypts the message and ensures the validity of the signature by checking whether it comes from the user in the `fromId` field in the message.

Long term, document the details of both the broadcast and peer-to-peer channels.

21. Queue InsertBefore implementation does not check for nil

Severity: Informational
Type: Data Validation
Target: `internal/common/list.go`

Difficulty: High
Finding ID: TOB-ATSS-021

Description

The `common/list.go` file implements a queue data structure with a mutex. The last function in the file, `InsertBefore`, inserts a value before the element `e`. The implementation uses `list.InsertBefore`, which performs the following, according to the documentation:

> *InsertBefore inserts a new element e with value v immediately before mark and returns e. If mark is not an element of l, the list is not modified. The mark must not be nil.*

To prevent a null dereference in the call to `list.InsertBefore`, the system needs to check that the element `e` is not null before the call.

```
// InsertBefore insert value before element e
func (q *Queue) InsertBefore(v interface{}, e *list.Element) {
        q.m.Lock()
        defer q.m.Unlock()
        q.l.InsertBefore(v, e)
}
```

Figure 21.1: `internal/common/list.go#L70-L75`

Exploit Scenario

An attacker triggers a function that calls `InsertBefore` with a `nil` element `e`, leading to a program crash.

Recommendations

Short term, add a check in the implementation that verifies that `e` is not `nil`, and ensure that the system falls back if it is.

References

- Go Documentation: func (*list) InsertBefore

22. Nonexistent modular inverse leads to crash

Severity: Medium
Difficulty: Medium
Type: Cryptography
Finding ID: TOB-ATSS-022
Target: `smpc-lib/ecdsa/signing/round_7.go`

Description
During round 7 of the signature protocol, participants add together each of the `delta`$_i$ values received from the other participants and compute the modular inverse of that sum. This inverse is then used in a scalar multiplication operation.

```
deltaSumInverse := new(big.Int).ModInverse(round.temp.deltaSum, secp256k1.S256().N)
deltaGammaGx, deltaGammaGy := secp256k1.S256().ScalarMult(GammaGSumx, GammaGSumy, deltaSumInverse.Bytes())
```

Figure 22.1: `smpc-lib/ecdsa/signing/round_7.go#L70-L71`

However, the `deltaSumInverse` variable is never checked against `nil`; this variable will be `nil` if `deltaSum` does not have a modular inverse modulo the curve order.

Exploit Scenario
An attacker crashes other nodes by waiting to receive all `delta`$_i$ values from the other participants and then sending `-sum(delta`$_i$`)` as his `delta`$_i$ value, causing `deltaSum` to zero, which does not have a modular inverse.

Recommendations
Short term, add a check to verify that `deltaSumInverse` is not `nil`, and ensure that the system falls back if it is.

23. Modularly equal nodeIDs cause crashes when signing and resharing

Severity: Medium
Difficulty: Low
Type: Cryptography
Finding ID: TOB-ATSS-023
Target: `smpc-lib/ecdsa/reshare/round_1.go`, `smpc-lib/ecdsa/signing/round_1.go`, `smpc-lib/eddsa/signing/round_4.go`, `smpc-lib/crypto/ed/vss.go`, `smpc-lib/crypto/ec2/vss2.go`

Description

Participants compute a Lagrangian coefficient in both the signature and reshare protocols. This coefficient is the product over `i` of $\mathtt{ID}_i$ `/ (`$\mathtt{ID}_i$ `- SelfID)`, in which the division is computed by multiplying with the modular inverse. If a user's $\mathtt{ID}_i$ is modularly equal to the current party's `SelfID`, the subtraction will be modularly equal to zero, and the `ModInverse` will return `nil`. If this occurs, the code will crash in the subsequent call to `Mul(subInverse, v)`.

```
for k, v := range round.idreshare {
	if k == index {
		continue
	}

	sub := new(big.Int).Sub(v, self)
	subInverse := new(big.Int).ModInverse(sub, secp256k1.S256().N)
	times := new(big.Int).Mul(subInverse, v)
	lambda1 = new(big.Int).Mul(lambda1, times)
	lambda1 = new(big.Int).Mod(lambda1, secp256k1.S256().N)
}
```

Figure 23.1: `smpc-lib/ecdsa/reshare/round_1.go#L61-L71`

The same issue occurs in the signing protocol and in the Feldman VSS implementation:

```
for k, v := range round.idsign {
	if k == curIndex {
		continue
	}

	sub := new(big.Int).Sub(v, self)
	subInverse := new(big.Int).ModInverse(sub, secp256k1.S256().N)
	times := new(big.Int).Mul(subInverse, v)
	lambda1 = new(big.Int).Mul(lambda1, times)
	lambda1 = new(big.Int).Mod(lambda1, secp256k1.S256().N)
}
```

Figure 23.2: `smpc-lib/ecdsa/signing/round_1.go#L68-L78`

```
for kk, vv := range round.idsign {
	if kk == curIndex {
		continue
	}
	var indexByte [32]byte
	copy(indexByte[:], vv.Bytes())

	var time [32]byte
	t := indexByte //round.temp.uids[oldindex]
	tt := curByte  //round.temp.uids[cur_oldindex]
	ed.ScSub(&time, &t, &tt)
	time = ed.ScModInverse(time, order)
	ed.ScMul(&time, &time, &t)
	ed.ScMul(&lambda, &lambda, &time)
}
```

Figure 23.3: `smpc-lib/eddsa/signing/round_4.go#L119-L134`

```
for i := 0; i < len(shares); i++ {
	if j != i {
		var time [32]byte
		ScSub(&time, &ids[i], &ids[j])
		time = ScModInverse(time, order)

		ScMul(&time, &time, &ids[i])

		ScMul(&times, &times, &time)
	}
}
```

Figure 23.4: `smpc-lib/crypto/ed/vss.go#L149-L159`

```
for j := 0; j < len(xSet); j++ {
	if j != i {
		sub := new(big.Int).Sub(xSet[j], share.ID)
		subInverse := new(big.Int).ModInverse(sub, s256.S256().N)
		div := new(big.Int).Mul(xSet[j], subInverse)
		times = new(big.Int).Mul(times, div)
		times = new(big.Int).Mod(times, s256.S256().N)
	}
}
```

Figure 23.5: `smpc-lib/crypto/ec2/vss2.go#L128-L136`

When parsing the node's ID, participants check for repetitions, but this is not performed modulo the order of the curve.

```
for _, enode := range enodes {
	node, err := discover.ParseNode(enode)
	if err != nil {
		msg := fmt.Sprintf("CheckAddPeer, parse err enode: %v", enode)
		return thshall, errors.New(msg)
	}
	if nodeid[node.ID] == 1 {
		msg := fmt.Sprintf("CheckAddPeer, enode: %v, err: repeated", enode)
		return thshall, errors.New(msg)
	}
	nodeid[node.ID] = 1
	...
```

Figure 23.6: The code checking for repetitions in the nodes IDs, but not modularly (`p2p/layer2/smpc.go#L417-L427`)

Exploit Scenario
An attacker chooses an ID that is modularly equal to another player, leading to a crash when that player computes and uses the Lagrangian coefficient.

Recommendations
Short term, check all uses of modular inverse functions and their return values for errors. Additionally, include a verification step to guarantee that all participants have modularly different node IDs.

24. EdDSA protocol ignores final signature verification

Severity: Medium
Difficulty: Medium
Type: Cryptography
Finding ID: TOB-ATSS-024
Target: `smpc-lib/eddsa/signing/round_7.go`

Description

During round 7 of the EdDSA signature protocol, participants receive partial signatures from each party. These partial signatures are added together to form the final group signature on a message. Once each party generates the signature, all parties verify this signature against the group's message and public key to ensure that the signature is valid. However, the implementation ignores the result of the verification and saves the signature regardless of whether it was valid.

```
inputVerify := InputVerify{FinalR: round.temp.FinalRBytes, FinalS: FinalS, Message:
[]byte(round.temp.message), FinalPk: round.temp.pkfinal}

var pass = EdVerify(inputVerify)
fmt.Printf("===========ed verify, pass = %v============\n", pass)

//r
rx := hex.EncodeToString(round.temp.FinalRBytes[:])
sx := hex.EncodeToString(FinalS[:])
fmt.Printf("===========ed sign, round7.start, rx = %v, sx = %v============\n", rx, sx)

//////test
signature := new([64]byte)
copy(signature[:], round.temp.FinalRBytes[:])
copy(signature[32:], FinalS[:])

fmt.Printf("================= ed sign 25519,sig = %v, pk = %v, msg = %v, sig str = %v, pk
str = %v, msg str = %v ======================\n", signature, round.temp.pkfinal,
round.temp.message, hex.EncodeToString(signature[:]),
hex.EncodeToString(round.temp.pkfinal[:]), hex.EncodeToString(round.temp.message[:]))
suss := ed25519.Verify(&round.temp.pkfinal, []byte(round.temp.message), signature)
fmt.Printf("===========ed verify, success = %v============\n", suss)

/////////solana
/*suss = edlib.Verify(round.temp.pkfinal[:],round.temp.message,signature[:])
fmt.Printf("===========ed lib verify, success = %v============\n",suss)

suss = Verify(round.temp.pkfinal[:],round.temp.message,signature[:])
fmt.Printf("===========ed lib at local verify, success = %v============\n",suss)*/
/////////solana

round.end <- EdSignData{Rx: round.temp.FinalRBytes, Sx: FinalS}
```

Figure 24.1: `smpc-lib/eddsa/signing/round_7.go#L50-L77`

This final verification is important for the security of the protocol. Without this check, a malicious party could corrupt her partial signature in such a way that only she can reconstruct the actual group signature.

Exploit Scenario
An attacker intentionally corrupts her partial signature so that every group will reconstruct an incorrect signature, except for her. The parties do not detect any issues because the results of their verifications are not checked.

Recommendations
Short term, adjust the implementation in round 7 so that each party returns an error if the EdDSA signature is not properly verified.

25. Weak Fiat-Shamir transformation in zero-knowledge proof of phase 6

Severity: High
Difficulty: Low
Type: Cryptography
Finding ID: TOB-ATSS-025
Target: `smpc-lib/crypto/ec2/stZK.go`

Description
The commit `036df7c99b07d422936` implements a zero-knowledge proof required for phase 6 of the threshold signature protocol GG20. Zero-knowledge proofs are essential to the security guarantees of the protocol, ensuring that the parties remain honest. The S_i and T_i zero-knowledge proofs are both used in the signing protocol and assure a verifier that the prover has knowledge of the two exponents σ and l such that $S = R^\sigma$ and $T = g^\sigma h^l$, where S, R, and T are known to the verifier of the proof.

The implementation follows the Fiat-Shamir transformation and uses a hash function to non-interactively generate the verifier challenge `e`.

```
e := Sha512_256i(T1X, T1Y, hGx, hGy, Gx, Gy, alphax, alphay, betaX, betaY)
```

Figure 25.1: `smpc-lib/crypto/ec2/stZK.go#L54`

However, the calculation does not include all public parameters: `S` and `R` are missing.

Exploit Scenario
An attacker notices that the hash function does not include all parameters and is able to forge proofs, undermining the security guarantees of the protocol.

Recommendations
Short term, include the public values `S` and `R` in the computation of the hash function.

Long term, review all utilizations of the Fiat-Shamir transformation for missing public parameters.

26. Nonbinding Pedersen commitments

Severity: High
Difficulty: Low
Type: Cryptography
Finding ID: TOB-ATSS-026
Target: `smpc-lib/ecdsa/signing/{round5.go, round_9.go}`

Description

The commits `036df7c99b07d422936` and `82f5cd677a989c05c5` implement zero-knowledge proofs required for phases 3 and 6 of the threshold signature protocol GG20. These proofs assure a verifier that the prover has knowledge of the two exponents σ and `l` such that `T` = $g^{\sigma}h^{l}$, where `T` is known to the verifier. Values g and h are distinct public generators for the group with an unknown discrete-logarithm relationship.

In the implementation, the base generator for the underlying curve is used for both g and h. As a result, an attacker can forge different σ' and `l'` exponents that commit to the same value, breaking the scheme.

```
// gg20: calculate T_i = g^sigma_i * h^l_i = sigma_i*G + l_i*h*G
l1 := random.GetRandomIntFromZn(secp256k1.S256().N)
one,_ := new(big.Int).SetString("1",10)
Gx,Gy := secp256k1.S256().ScalarBaseMult(one.Bytes())
l1Gx,l1Gy := secp256k1.S256().ScalarMult(Gx,Gy,l1.Bytes())
sigmaGx,sigmaGy := secp256k1.S256().ScalarBaseMult(sigma1.Bytes())
t1X,t1Y := secp256k1.S256().Add(sigmaGx,sigmaGy,l1Gx,l1Gy)
// gg20: generate the ZK proof of T_i
tProof := ec2.TProve(t1X,t1Y,Gx,Gy,sigma1,l1)
```

Figure 26.1: `smpc-lib/ecdsa/signing/round_5.go#L112-L120`

```
one, _ := new(big.Int).SetString("1", 10)
Gx, Gy := secp256k1.S256().ScalarBaseMult(one.Bytes())

var s1x *big.Int
var s1y *big.Int

for k := range round.idsign {
       msg8, _ := round.temp.signRound8Messages[k].(*SignRound8Message)
       msg5, _ := round.temp.signRound5Messages[k].(*SignRound5Message)
       if ok := ec2.STVerify(msg8.S1X, msg8.S1Y, msg5.T1X, msg5.T1Y,
round.temp.deltaGammaGx, round.temp.deltaGammaGy, Gx, Gy, msg8.STpf);
```

Figure 26.2: `smpc-lib/ecdsa/signing/round_9.go#L40-L49`

Exploit Scenario

An attacker notices that the proof is nonbinding and is able to reveal a different commit than the one he previously bound.

Recommendations

Short term, modify the system so that it creates a new generator in a verifiable way. One way to do this is to hash the coordinates of the base generator with a counter until this hash is a valid x-coordinate for a point in the elliptic curve.

27. Insufficient input validation in the zero-knowledge proofs of phase 3 and phase 6

Severity: High
Type: Cryptography
Target: `smpc-lib/ecdsa/signing/{round5.go`, `round_9.go}`

Difficulty: Low
Finding ID: TOB-ATSS-027

Description
The commits `036df7c99b07d422936` and `82f5cd677a989c05c5` implement zero-knowledge proofs required for phases 3 and 6 of the threshold signature protocol GG20. These proofs assure a verifier that the prover has knowledge of the two exponents σ and `l` such that `T` = $g^{\sigma}h^{l}$, where `T` is known to the verifier. Values `g` and `h` are distinct public generators for the group with an unknown discrete-logarithm relationship.

When the verifier checks the validity of the proof, he does not validate the input parameters against potentially malicious values. The verifier should ensure that all of the input values are nonzero and not equal to one (and these checks need to be performed modularly). Currently, a malicious prover could set the values T and `alpha` to the point at infinity and `proof.T` and `proof.U` to zero, and the proof for `T` will pass verification.

Exploit Scenario
An attacker notices that the proof verification does not validate inputs and is able to bypass it by providing the appropriate parameters.

Recommendations
Short term, validate all input from the proof statement to be modularly different from zero or one (in the case of a scalar) and different from the point at infinity (in the case of a point of an elliptic curve).

A. Vulnerability Classifications

Vulnerability Classes	
Class	**Description**
Access Controls	Related to authorization of users and assessment of rights
Auditing and Logging	Related to auditing of actions or logging of problems
Authentication	Related to the identification of users
Configuration	Related to security configurations of servers, devices, or software
Cryptography	Related to protecting the privacy or integrity of data
Data Exposure	Related to unintended exposure of sensitive information
Data Validation	Related to improper reliance on the structure or values of data
Denial of Service	Related to causing a system failure
Error Reporting	Related to the reporting of error conditions in a secure fashion
Patching	Related to keeping software up to date
Session Management	Related to the identification of authenticated users
Timing	Related to race conditions, locking, or the order of operations
Undefined Behavior	Related to undefined behavior triggered by the program

Severity Categories	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices or Defense in Depth.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is relatively small or is not a risk the customer has indicated is important.
Medium	Individual users' information is at risk; exploitation could pose

	reputational, legal, or moderate financial risks to the client.
High	The issue could affect numerous users and have serious reputational, legal, or financial implications for the client.

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

B. Code Maturity Classifications

Code Maturity Classes	
Category Name	**Description**
Access Controls	Related to the authentication and authorization of components
Arithmetic	Related to the proper use of mathematical operations and semantics
Assembly Use	Related to the use of inline assembly
Centralization	Related to the existence of a single point of failure
Upgradeability	Related to contract upgradeability
Function Composition	Related to separation of the logic into functions with clear purposes
Front-Running	Related to resilience against front-running
Key Management	Related to the existence of proper procedures for key generation, distribution, and access
Monitoring	Related to the use of events and monitoring procedures
Specification	Related to the expected codebase documentation
Testing and Verification	Related to the use of testing techniques (unit tests, fuzzing, symbolic execution, etc.)

Rating Criteria	
Rating	**Description**
Strong	The component was reviewed, and no concerns were found.
Satisfactory	The component had only minor issues.
Moderate	The component had some issues.
Weak	The component led to multiple issues; more issues might be present.
Missing	The component was missing.

Not Applicable	The component is not applicable.
Not Considered	The component was not reviewed.
Further Investigation Required	The component requires further investigation.

C. Code-Quality Findings

The following findings are not associated with specific vulnerabilities. However, they enhance code readability and may prevent the introduction of vulnerabilities in the future.

- In various locations, the value 2048 is a magic number hard-coded for Paillier key sizes. Consider replacing all of these with a single constant.
- Several locations use the pattern `if a == b { return true; }` instead of `return a == b`:
  - `smpc-lib/crypto/ec2/MtAZK1_nhh.go` line 126
  - `smpc-lib/crypto/ec2/MtAZK2_nhh.go` line 167
  - `smpc-lib/crypto/ec2/MtAZK3_nhh.go` line 177
  - `smpc-lib/crypto/ec2/paillier.go` line 189
  - `smpc-lib/crypto/ec2/schnorrZK.go` line 91
  - `smpc-lib/crypto/ed/commit.go` line 79
  - `smpc-lib/crypto/ed/schnorrZK.go` line 101
  - `smpc-lib/crypto/ed/vss.go` line 129
  - `smpc/api_reqsign.go` lines 638 and 652
- Several locations have unused variables and fields:
  - `smpc-lib/ecdsa/keygen/local_dnode.go` line 49: `kgRound7Messages` is unused.
  - `smpc-lib/ecdsa/reshare/round_0.go` line 28: `zero` is unused.
  - `smpc-lib/ecdsa/signing/round_1.go` line 31: `zero` is unused.
  - `smpc-lib/eddsa/keygen/round_0.go` line 27: `zero` is unused.
  - `smpc-lib/eddsa/signing/round_1.go` line 31: `zero` is unused.
  - `smpc-lib/smpc/dnode.go`:
    - line 46: `mtx` is unused.
    - line 57: `(*BaseDNode).setRound` is unused.
    - line 70: `(*BaseDNode).advance` is unused.
    - line 74: `(*BaseDNode).lock` is unused.
    - line 78: `(*BaseDNode).unlock` is unused.
  - `smpc/api_reqsign.go` line 55: This value of `exist` is never used.
  - `smpc/pre_signdata.go` line 1006: This value of `old` is never used.
  - `smpc/reqaddr.go`:
    - line 41: `reqdataTrytimes` is unused.
    - line 44: `reqdataTimeout` is unused.
- The ECDSA key generation protocol also generates a BIP-32 private key but does not document how this key will be used.
- Closing resources or connections in a loop with `defer` causes the resources to be released only when the function terminates. Consider closing the connection without `defer` in `p2p/discv5/ntp.go#L76-L82` and `p2p/discover/ntp.go#L73-L79`.

- A large code block is never reached in `p2p/discover/group.go#L926-L959` since the `if` statement returns in both the `if` and the `else` clause.
- In `internal/common/path.go#L41`, `os.IsNotExist(err)` should be replaced with `errors.Is(err, fs.ErrNotExist)`. See the documentation of [`IsNotExist`]().
- In `smpc-lib/ecdsa/keygen/round_2.go#L55-L76` and `smpc-lib/ecdsa/reshare/round_2.go#L78-L101`, you can delay the creation of the `kg` object until the check that it is necessary. This prevents unnecessary resource consumption.
- In `smpc-lib/crypto/ec2/paillier.go`, the `ZkFactProve` and `ZkFactVerify` functions are used only in testing. Consider removing unused code or moving the functions to the test files.
- In `rpc/smpc/rpc.go#L758-L761`, `Server.stop()` never executes since reading from the channel `exit` will block it forever.
- There is a potential null dereference due to unchecked return errors in the following:
    - `p2p/layer2/smpc.go#L338`
    - `p2p/discv5/net.go#L733-L736`
- Integers are parsed as a larger size (64 and 32) but then used with a smaller typed integer (32 and 16 bit) in the following:
    - `p2p/simulations/adapters/types.go#L200-L204`
    - `cmd/gsmpc-client/main.go#L127-L144`
- An integer is parsed as an unsigned `int` but used as a signed integer in `p2p/simulations/http.go#L496-L500`.
- The `fmt.Printf` function has more arguments than formats in `rpc/smpc/p2p.go#L241.`
- In `rpc/smpc/p2p.go#L209-L223`, there is an unnecessary iteration over the hashmap to find a key. Instead, you should access the desired key directly.
- Function signatures are more readable when repeated types are written once: `func NewSTProof(T1X *big.Int,T1Y *big.Int,Rx *big.Int,Ry *big.Int,hGx *big.Int,hGy *big.Int,sigma1 *big.Int,l1 *big.Int) *STProof` can become `func NewSTProof(T1X, T1Y, Rx, Ry, hGx, hGy, sigma1, l1 *big.Int) *STProof`.

D. Fix Log

After the assessment, AnySwap informed Trail of Bits that it had addressed issues identified in the audit through various pull requests. The audit team verified each fix to ensure that it would appropriately address the corresponding issue. The results of this audit are provided in the table below.

ID	Title	Severity	Status
1	Weak Fiat-Shamir transformation in Schnorr's zero-knowledge proof	High	Fixed
2	VSS generates shares from ID coordinates	High	Fixed
3	Unhandled errors in cryptographically sensitive operations	High	Fixed
4	MtA proof verification always rejects the upper bound q^3	Informational	Fixed
5	ECDSA signature verification can panic	Medium	Fixed
6	Missing checks in NtildeProof verification could lead to Golden Shoe attack	High	Fixed
7	Insufficient input validation in VSS operations could result in trivial or unrecoverable shares	Low	Fixed
8	Missing domain separation in various hash computations could result in canonicalization attacks	High	Fixed
9	Weak Fiat-Shamir transformation in MtA proofs	High	Fixed
10	Nonbinding "MtA with check" proof	Medium	Fixed
11	Missing implementation of phase 5 of the signing protocol	Medium	Fixed
12	Paillier modulus is not proved to be square-free	Medium	Fixed
13	Unnecessary key generation in the reshare protocol	Informational	Fixed
14	Fiat-Shamir challenge in MtA proofs use the wrong modulus	Informational	Fixed

15	MtA proof verification implicitly trusts the prover's Paillier public key	High	Fixed
16	Insufficient input validation in MtA verify functions could allow proof forgery	Low	Fixed
17	Key generation includes the wrong Schnorr proof	Informational	Fixed
18	Incoming messages are not validated and could cause crashes	Medium	Fixed
19	In the transport layer, broadcast messages are not signed	High	Fixed
20	In the transport layer, peer-to-peer messages are not signed	High	Fixed
21	Queue InsertBefore implementation does not check for nil	Informational	Fixed
22	Nonexistent modular inverse leads to crash	Medium	Fixed
23	Modularly equal nodeIDs cause crashes when signing and resharing	Medium	Fixed
24	EdDSA protocol ignores final signature verification	Medium	Fixed
25	Weak Fiat-Shamir transformation in zero-knowledge proof of phase 6	High	Fixed
26	Nonbinding Pedersen commitments	High	Fixed
27	Insufficient input validation in the zero-knowledge proofs of phase 3 and phase 6	High	Fixed

For additional information on each fix, please refer to the Detailed Fix Log on the following page.

Detailed Fix Log

TOB-ATSS-001: Weak Fiat-Shamir transformation in Schnorr's zero-knowledge proof
Fixed. The public value g^{sk} is now included in the computation of the hash function, but domain separation is not used.

TOB-ATSS-002: VSS generates shares from ID coordinates
Fixed. A check has been added to guarantee that the user's ID is nonzero modulo the curve of the protocol.

TOB-ATSS-003: Unhandled errors in cryptographically sensitive operations
Fixed. The codebase has been updated to catch several previously uncaught errors.

TOB-ATSS-004: MtA proof verification always rejects the upper bound q^3
Fixed. The comparison has been fixed so that the check does not reject the upper bound.

TOB-ATSS-005: ECDSA signature verification can panic
Fixed. Checks have been added to verify that `(r, s)` is in the interval `[1, q-1]`, that the provided public key is on the correct elliptic curve, and that the public key is not the point at infinity.

TOB-ATSS-006: Missing checks in NtildeProof verification could lead to Golden Shoe attack
Fixed. The recommended checks for h_1 and h_2 have been added, and the prover now includes a proof that Ñ is the product of two primes.

TOB-ATSS-007: Insufficient input validation in Feldman VSS operations could result in trivial or unrecoverable shares
Fixed. Checks have been added to prevent trivial or unrecoverable shares.

TOB-ATSS-008: Missing domain separation in various hash computations could result in canonicalization attacks
Fixed. The examples mentioned in issue TOB-ATSS-008 now include domain separation.

TOB-ATSS-009: Weak Fiat-Shamir transformation in MtA proofs
Fixed. The public statement values have been included in the computation of the hash function.

TOB-ATSS-010: Nonbinding "MtA with check" proof
Fixed. The MtA implementation has been adjusted to include the discrete logarithm statement.

TOB-ATSS-011: Missing implementation of phase 5 of the signing protocol
Fixed. Phases 5 and 6 of the signing protocol have been implemented.

TOB-ATSS-012: Paillier modulus is not proved to be square-free
Fixed. A zero-knowledge proof that E_i is a square-free integer has been added.

TOB-ATSS-013: Unnecessary key generation in the reshare protocol
Fixed. The reshare protocol has been updated so that only new party members generate new Paillier keys and Fujisaki-Okamoto commitments.

TOB-ATSS-014: Fiat-Shamir challenge in MtA proofs use the wrong modulus
Fixed. The Fiat-Shamir challenge now uses the correct modulus.

TOB-ATSS-015: MtA proof verification implicitly trusts the prover's Paillier public key
Fixed. The verifiers now reconstruct the `N2` and `G` values rather than using the values submitted by the prover.

TOB-ATSS-016: Insufficient input validation in MtA verify functions could allow proof forgery
Fixed. All of the recommended checks have been added.

TOB-ATSS-017: Key generation includes wrong Schnorr proof
Fixed. The proof of knowledge of u_i has been replaced with the proof of knowledge of x_i.

TOB-ATSS-018: Incoming messages are not validated and could cause crashes
Fixed. Incoming messages are now checked to ensure they contain the required fields.

TOB-ATSS-019: In the transport layer, broadcast messages are not signed
Fixed. Checks have been added to ensure participants sign their broadcast messages.

TOB-ATSS-020: In the transport layer, peer-to-peer messages are not signed
Fixed. Checks have been added to ensure participants sign their peer-to-peer messages.

TOB-ATSS-021: Queue InsertBefore implementation does not check for nil
Fixed. A check has been added to verify that `e` is not `nil`.

TOB-ATSS-022: Nonexistent modular inverse leads to crash
Fixed. A check has been added to ensure the result of `ModInverse` is not `nil`.

TOB-ATSS-023: Modularly equal nodeIDs cause crashes when signing and resharing
Fixed. Checks have been added to the uses of the modular inverse function to catch errors.

TOB-ATSS-024: EdDSA protocol ignores final signature verification

Fixed. The implementation returns an error if the EdDSA signature is not properly verified.

TOB-ATSS-025: Weak Fiat-Shamir transformation in zero-knowledge proof of phase 6
Fixed. The public values `S` and `R` are now included in the computation of the hash function.

TOB-ATSS-026: Nonbinding Pedersen commitments
Fixed. The generator `h` is now computed so that its discrete-logarithm relationship with `g` is unknown.

TOB-ATSS-027: Insufficient input validation in the zero-knowledge proofs of phase 3 and phase 6
Fixed. Checks have been added to ensure that the proof scalars are not modularly equal to zero or one and that the components of the proof that are elliptic curve points are not the point at infinity.