Audit of HTSS

AMIS Technologies Co., Ltd.

03 November 2022
Version: 1.2

Presented by:
Kudelski Security Research Team
Kudelski Security – Nagravision Sarl

Corporate Headquarters
Kudelski Security – Nagravision Sarl
Route de Genève, 22-24
1033 Cheseaux sur Lausanne
Switzerland

DOCUMENT PROPERTIES

Version:	1.2
File Name:	Audit_HTSS
Publication Date:	03 November 2022
Confidentiality Level:	Restricted
Document Owner:	Tommaso Gagliardoni
Document Recipient:	Anderson Lin
Document Status:	Approved

TABLE OF CONTENTS

EXECUTIVE SUMMARY 5
1.1 Engagement Scope 5
1.2 Engagement Analysis 5
1.3 Observations 6
1.4 Issue Summary List 7
2. METHODOLOGY 8
2.1 Kickoff 8
2.2 Ramp-up 8
2.3 Review 8
2.4 Reporting 9
2.5 Verify 10
2.6 Additional Note 10
3. TECHNICAL DETAILS OF SECURITY FINDINGS (FOR FROST) 11
3.1 Party's public keys only computed during signing 11
3.2 Issues in identity-commitment construction 12
3.3 Missing nonzero checks 13
4. TECHNICAL DETAILS OF SECURITY FINDINGS (FOR CGGMP) 14
4.1 Key refreshing generates a set of shares of zero 14
4.2 Messages sent as P2P instead of broadcast 15
4.3 ZKP Proofs sent before collecting all responses 16
4.4 Wrong modulo reduction 18
5. OTHER OBSERVATIONS 19
5.1 Protocol steps interleaved within different files 19
5.2 Use of notation from outdated versions of paper 20
5.3 Limited testing for key refresh 21
APPENDIX A: ABOUT KUDELSKI SECURITY 22
APPENDIX B: DOCUMENT HISTORY 23
APPENDIX C: SEVERITY RATING DEFINITIONS 24

TABLE OF FIGURES

Figure 1 Issue Severity Distribution 6
Figure 2 Methodology Flow 8

EXECUTIVE SUMMARY

Kudelski Security ("Kudelski", "we"), the cybersecurity division of the Kudelski Group, was engaged by AMIS Technologies Co., Ltd. ("the Client") to conduct an external security assessment in the form of a code audit of a major feature update of the hierarchical threshold signature scheme (HTSS) library Alice ("the Product") developed by the Client.

The assessment was conducted remotely by the Kudelski Cybersecurity Research Team. The audit took place in August and September 2022, and focused on the following objectives:

- To provide a professional opinion on the maturity, adequacy, and efficiency of the software solution in exam.
- To check compliance with existing standards.
- To identify potential security or interoperability issues and include improvement recommendations based on the result of our analysis.

This report summarizes the analysis performed and findings. It also contains detailed descriptions of the discovered vulnerabilities and recommendations for remediation.

1.1 Engagement Scope

The scope of the audit was a cryptographic review of the Client's implementation in Golang of two TSS schemes:

1) CGGMP for threshold ECDSA with support for hierarchical shares:
   Paper: https://eprint.iacr.org/2021/060
   Repository: https://github.com/getamis/alice/tree/master/crypto/tss/ecdsa/cggmp

2) FROST for threshold EdDSA with support for hierarchical shares:
   Paper: https://eprint.iacr.org/2020/852
   Repository: https://github.com/getamis/alice/tree/master/crypto/tss/eddsa/frost

The original commit number was `f2af6dd139b4ab9b97b28356583854423ac4af73`.

1.2 Engagement Analysis

The engagement consisted of a ramp-up phase where the necessary documentation about the technological standards and design of the solution in exam was acquired, followed by a manual inspection of the code provided by the Client and the drafting of this report.

As a result of our work, we identified **3 High**, **2 Medium**, **2 Low**, and **3 Informational** findings.

Most of these findings and observations are related to deviations from the original protocols. We observe that the related academic papers are very recent and ripe with typos and mistakes, so a certain difficulty in implementing them correctly has to be expected.

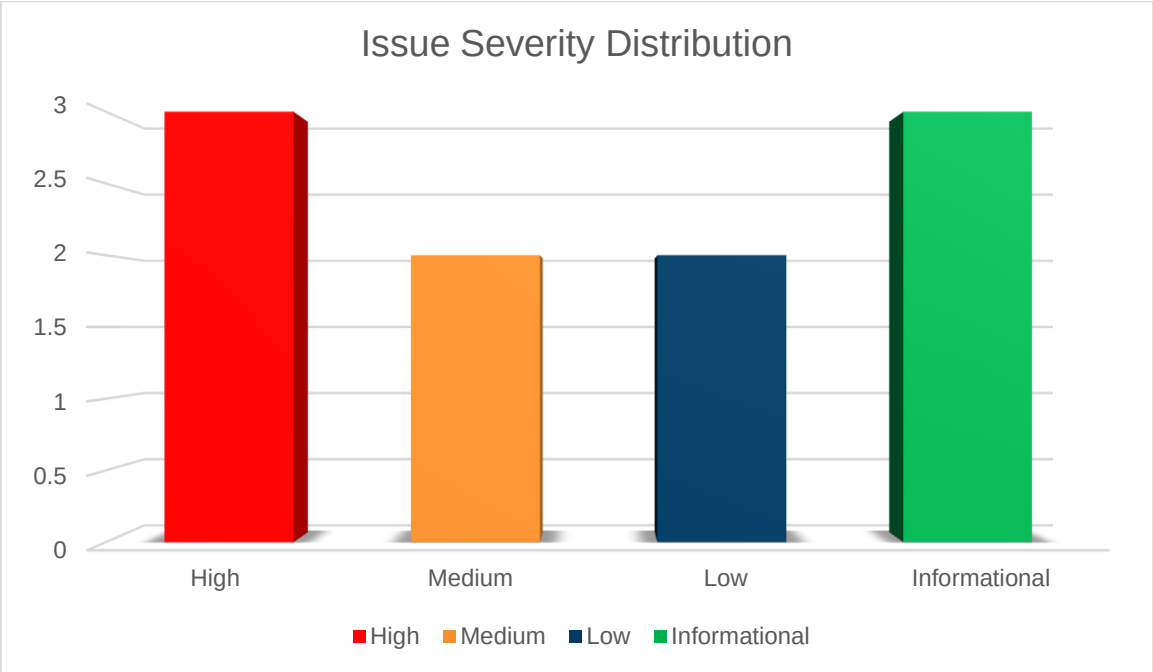


Figure 1 Issue Severity Distribution

1.3 Observations

The Product provides an implementation of different TSS schemes, originally GG18 for ECDSA, by adding support for hierarchical shares, thereby turning these schemes into HTSS. The Product has been updated by adding support for two new schemes, CGGMP for ECDSA and FROST for EdDSA. Regarding CGGMP, part of the code and of the low-level functionalities related to hierarchical shares implementation are borrowed from the GG18 code, which was already audited in the past, so we did not reaudit that part. Regarding FROST, its support is still experimental, and the distributed key generation algorithm (DKG) is currently borrowed from the GG18 code. For both schemes, timing side-channel attacks were considered not in scope.

In general, we found the implementation to be of high standard and we believe that all the identified vulnerabilities can be easily addressed. Moreover, we did not find evidence of any hidden backdoor or malicious intent in the code.

1.4 Issue Summary List

The following security issues were found (for FROST):

ID	SEVERITY	FINDING	STATUS
KS-AMC-F-01	High	Party's public keys only computed during signing	Remediated
KS-AMC-F-02	Medium	Issues in identity-commitment construction	Remediated
KS-AMC-F-03	Low	Missing nonzero checks	Remediated

The following security issues were found (for CGGMP):

ID	SEVERITY	FINDING	STATUS
KS-AMC-F-04	High	Key refreshing generates a set of shares of zero	Remediated
KS-AMC-F-05	High	Messages sent as P2P instead of broadcast	Remediated
KS-AMC-F-06	Medium	ZKP Proofs sent before collecting all responses	Remediated
KS-AMC-F-07	Low	Wrong modulo reduction	Remediated

The following are non-security observations related to general design and optimization:

ID	SEVERITY	FINDING	STATUS
KS-AMC-O-01	Informational	Protocol steps interleaved within different files	Informational
KS- AMC-O-02	Informational	Use of notation from outdated versions of paper	Remediated
KS- AMC-O-03	Informational	Limited testing for key refresh	Remediated

2. METHODOLOGY

For this engagement, Kudelski used a methodology that is described at high-level in this section. This is broken up into the following phases.

Figure 2 Methodology Flow

2.1 Kickoff

The project was kicked off when all the sales activities had been concluded. We set up a kickoff meeting where project stakeholders were gathered to discuss the project as well as the responsibilities of participants. During this meeting we verified the scope of the engagement and discussed the project activities. It was an opportunity for both sides to ask questions and get to know each other. By the end of the kickoff there was an understanding of the following:

- Designated points of contact
- Communication methods and frequency
- Shared documentation
- Code and/or any other artifacts necessary for project success
- Follow-up meeting schedule, such as a technical walkthrough
- Understanding of timeline and duration

2.2 Ramp-up

Ramp-up consisted of the activities necessary to gain proficiency on the particular project. This included the steps needed for gaining familiarity with the codebase and technological innovations utilized, such as:

- Reviewing previous work in the area including academic papers
- Reviewing programming language constructs for the languages used in the code
- Researching common flaws and recent technological advancements

2.3 Review

The review phase is where most of the work on the engagement was performed. In this phase we analyzed the project for flaws and issues that could impact the security posture. This included an analysis of the architecture, a review of the code, and a specification matching to match the architecture to the implemented code.

In this code audit, we performed the following tasks:

1. Security analysis and architecture review of the original protocol
2. Review of the code written for the project

3. Assessment of the cryptographic primitives used
4. Compliance of the code with the provided technical documentation

The review for this project was performed using manual methods and utilizing the experience of the reviewer. No dynamic testing was performed, only the use of custom-built scripts and tools were used to assist the reviewer during the testing. We discuss our methodology in more detail in the following subsections.

Code Safety

We analyzed the provided code, checking for issues related to the following categories:

- General code safety and susceptibility to known issues
- Poor coding practices and unsafe behavior
- Leakage of secrets or other sensitive data through memory mismanagement
- Susceptibility to misuse and system errors
- Error management and logging

This is a general and not comprehensive list, meant only to give an understanding of the issues we have been looking for.

Cryptography

We analyzed the cryptographic primitives and components as well as their implementation. We checked in particular:

- Matching of the proper cryptographic primitives to the desired cryptographic functionality needed
- Security level of cryptographic primitives and their respective parameters (key lengths, etc.)
- Safety of the randomness generation in general as well as in the case of failure
- Safety of key management
- Assessment of proper security definitions and compliance to use cases
- Checking for known vulnerabilities in the primitives used

Technical Specification Matching

We analyzed the provided documentation and checked that the code matches the specification. We checked for things such as:

- Proper implementation of the documented protocol phases
- Proper error handling
- Adherence to the protocol logical description

2.4 Reporting

Kudelski delivered to the Client a preliminary report in PDF format that contained an executive summary, technical details, and observations about the project, which is also the general structure of the current final report.

The executive summary contains an overview of the engagement, including the number of findings as well as a statement about our general risk assessment of the project as a whole.

In the report we not only point out security issues identified but also informational findings for improvement categorized into several buckets:

- High
- Medium
- Low
- Informational

The technical details are aimed more at developers, describing the issues, the severity ranking and recommendations for mitigation.

As we performed the audit, we also identified issues that are not security related, but are general best practices and steps, that can be taken to lower the attack surface of the project.

As an optional step, we can agree on the creation of a public report that can be shared and distributed with a larger audience.

2.5 Verify

After the preliminary findings have been delivered, we verified the fixes applied by the Client. After these fixes were verified, we updated the status of the finding in the report.

The output of this phase was the current, final report with any mitigated findings noted.

2.6 Additional Note

It is important to notice that, although we did our best in our analysis, no code audit assessment is per se guarantee of absence of vulnerabilities. Our effort was constrained by resource and time limits, along with the scope of the agreement.

In assessing the severity of some of the findings we identified, we kept in mind both the ease of exploitability and the potential damage caused by an exploit. Since this is a library, we ranked some of these vulnerabilities potentially higher than usual, as we expect the code to be reused across different applications with different input sanitization and parameters.

Correct memory management is left to Go and was therefore not in scope. Zeroization of secret values from memory is also not enforceable at a low level in a language such as Go.

While assessment the severity of the findings, we considered the impact, ease of exploitability, and the probability of attack. This is a solid baseline for severity determination. Information about the severity ratings can be found in **Appendix C** of this document.

3. TECHNICAL DETAILS OF SECURITY FINDINGS (FOR FROST)

This section contains the technical details of our findings as well as recommendations for mitigation.

3.1 Party's public keys only computed during signing

Finding ID: KS-AMC-F-01

Severity: High

Status: Remediated

Location: `eddsa/frost/signer/round_1.go:123`

Description and Impact Summary

In the code, each party's own public key value Y_i is computed during round 1 of signing, and transmitted to each other party to be used in round 2.

```
YPoint := ecpointgrouplaw.ScalarBaseMult(curve, share)
msgY, err := YPoint.ToEcPointMessage()
if err != nil {
    return nil, err
}
```

However, this is a deviation from the FROST protocol. In the paper, every party is responsible for reconstructing each other parties' public key using information exchanged during the key generation phase (Step 4 of Round 2 of Key Generation, Fig.1), and public keys of all participants are stored as an indicator of that party's identity.

> **4.** Each P_i calculates their public verification share $Y_i = g^{s_i}$, and the group's public key $Y = \prod_{j=1}^{n} \phi_{j0}$. Any participant can compute the public verification share of any other participant by calculating $Y_i = \prod_{j=1}^{n} \prod_{k=0}^{t-1} \phi_{jk}^{i^k \bmod q}$.

If this public key is first computed by each party during signing, instead, this gives no security guarantee because it implicitly assumes that every party behaves honestly at that step, by sending to everyone else their correctly computed public key.

Recommendation

We recommend checking and storing these values during key generation.

Status Details

This was fixed in PR #191 according to our recommendations. Now each party's Y_i is stored in the structure that defines the properties of the peer during key generation.

3.2 Issues in identity-commitment construction

Finding ID: KS-AMC-F-02

Severity: Medium

Status: Remediated

Location: `eddsa/frost/signer/round_1.go:329`

Description and Impact Summary

The function `computeB` fetches a pre-stored tuple comprising of a party's identity and stored commitments to their precomputed nonces. The resulting string is supposed to be unique and can be used through the signature generation. However, we found two problems.

The first one is that in one case, only the x coordinate of one point is used, while in the other case only the y one is used:

```
// Get xi,Di,Ei,.......
func computeB(x []byte, D, E *ecpointgrouplaw.ECPoint) []byte {
    var result []byte
    separationSign := []byte(",")
    result = append(result, x...)
    result = append(result, separationSign...)
    result = append(result, D.GetX().Bytes()...)
    result = append(result, separationSign...)
    result = append(result, E.GetY().Bytes()...)
    result = append(result, separationSign...)
    return result
}
```

The second one is that the corresponding fields are not fixed size and are just divided by a comma separator. When transforming this into a string, collisions can occur. For example, suppose that two point coordinates are represented as two byte-vectors of length 32 as:

`[0x00 A1 ... A31]` **`[B1 ... B32]`**

where **`B1`** is a byte representation of a comma. Now, suppose to have two other points:

`[A1 ... A31 B1]` **`[0x00 B2 ... B32]`**

After being processed by the function, in both cases we end up with the same bytestring:

`[<x> A1 ... A31 B1 B2 ... B32]`

therefore, subsequent collisions will occur.

Recommendation

We recommend performing a correct Edwards point representation using `ecpointEncoding` which also constrains the size of the fields to a fixed size.

Status Details

This was fixed in PR #188 according to our recommendations.

3.3 Missing nonzero checks

Finding ID: KS-AMC-F-03

Severity: Low

Status: Remediated

Location: `eddsa/frost/signer/round_1.go:299,371`

Description and Impact Summary

There are two modular reductions that are supposed to map to random elements of Z*, but since the output is not checked to be nonzero, there is a slight possibility that the result will end up in Z instead (zero element).

```
    h.Write(encodedPubKey[:])
    h.Write(message)
    digest := h.Sum(nil)
    result := new(big.Int).SetBytes(utils.ReverseByte(digest))
    return result.Mod(result, R.GetCurve().Params().N)
```

```
            temp, err = utils.HashProtosToInt(temp.Bytes(), &any.Any{
                Value: temp.Bytes(),
            }, &any.Any{
                Value: B,
            })
            tempMod = new(big.Int).Mod(temp, bit254)
```

At a very minimum, this formally breaks the correctness guarantees of the protocol.

Recommendation

Even if this only happens with negligible probability, due to the low-cost nature of the check we suggest adding it as a defense-in-depth mechanism.

Status Details

This was fixed in PR #190 according to our recommendations.

4. TECHNICAL DETAILS OF SECURITY FINDINGS (FOR CGGMP)

This section contains the technical details of our findings as well as recommendations for mitigation.

4.1 Key refreshing generates a set of shares of zero

Finding ID: KS-AMC-F-04

Severity: High

Status: Remediated

Location: `ecdsa/cggmp/refresh/round_3.go:194`

Description and Impact Summary

In the CGGMP paper, at the end of the protocols the peers finally compute their new share x_i^* and new partial public key X_k^* with the following:

> 2. When passing above verification for all $\mathcal{P}_j$, do:
>
> - Set $x_i^* = x_i + \sum_j x_j^i \mod q$.
> - Set $X_k^* = X_k \cdot \prod_j X_j^k$ for every k.

where x_i and X_k are the party's old secret share and the other parties' old partial public keys.

However, in the code implementation, the peers just set the new share to the sum of the decrypted shares received from other parties and its own refresh share, basically ending up with a new set of shares of 0. The same happens for the refresh of the partial public keys:

```
func (p *round3Handler) Finalize(logger log.Logger) (types.Handler, error) {
    curve := p.pubKey.GetCurve()
    refreshShare := new(big.Int).Set(p.refreshShare)
    sumpartialPubKey := pt.ScalarBaseMult(curve, p.refreshShare)
    partialPubKey := make(map[string]*pt.ECPoint)
    var err error
    for _, peer := range p.peers {
        plaintextShareBig := peer.round3.plaintextShareBig
        refreshShare = refreshShare.Add(refreshShare, plaintextShareBig)
        sumpartialPubKey, err = sumpartialPubKey.Add(pt.ScalarBaseMult(curve, plaintextShareBig))
        if err != nil {
            return nil, err
        }
    }
```

This means that after a refresh, or before any signature, the secret key is basically reset to a zero value.

Recommendation

We observe that the full CGGMP protocol is not used in the examples folder. The only test for the refresh protocol (`refresh_test.go`) first runs the protocol, but then adds back the previous shares from the DKG protocol in the test routine; it then proceeds to success verifying that the sum of all shares is equal to the old secret (global private key). So, the test routine actually implements the full protocol correctly, and this is probably why the issue has not been caught, but this is clearly not possible in the final implementation, as it is up to each party to manage and manipulate their own share. We conclude that the missing addition with the old share must be implemented in the `Finalize` routine.

Status Details

This was fixed in PR #203 according to our recommendations.

4.2 Messages sent as P2P instead of broadcast

Finding ID: KS-AMC-F-05

Severity: High

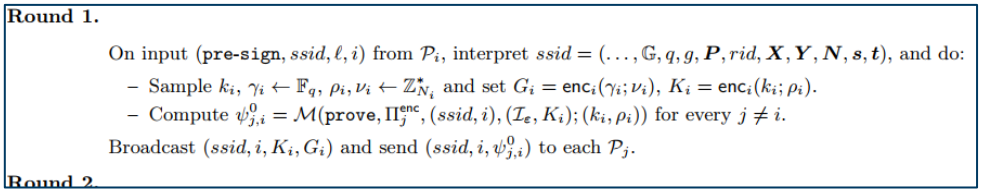
Status: Remediated

Location: `ecdsa/cggmp/sign/round_1.go:275`

`ecdsa/cggmp/signSix/round_1.go:311`

Description and Impact Summary

At the end of the first round of the presign phase, both in the 3-round and the 6-rounds variants (Fig. 7 and 9 of the paper, respectively), there is certain data that must be sent (privately, e.g., on a secure channel) to some other peers, and some data that must be reliably broadcast to all peers.



Round 1.

On input $(\texttt{pre-sign}, ssid, \ell, i)$ from $\mathcal{P}_i$, interpret $ssid = (\ldots, \mathbb{G}, q, g, \boldsymbol{P}, rid, \boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{N}, \boldsymbol{s}, \boldsymbol{t})$, and do:

- Sample $k_i, \gamma_i \leftarrow \mathbb{F}_q$, $\rho_i, \nu_i \leftarrow \mathbb{Z}^*_{N_i}$ and set $G_i = \mathsf{enc}_i(\gamma_i; \nu_i)$, $K_i = \mathsf{enc}_i(k_i; \rho_i)$.
- Compute $\psi^0_{j,i} = \mathcal{M}(\texttt{prove}, \Pi^{\mathsf{enc}}_j, (ssid, i), (\mathcal{I}_\varepsilon, K_i); (k_i, \rho_i))$ for every $j \neq i$.

Broadcast $(ssid, i, K_i, G_i)$ and send $(ssid, i, \psi^0_{j,i})$ to each $\mathcal{P}_j$.

Round 2.

However, in the code (`sendRound1Messages`) there is no such distinction, and all these messages are communicated peer-to-peer to every other party. This introduces potentially serious vulnerabilities, as there is no way to make sure that a malicious party is not sending the same “public” values to everyone else. In fact, a malicious party could “segment” the pool of other participants by sending a certain value to some of them, and another value to all the others. This is similar to the “forget-and-forgive” attack described for example in https://eprint.iacr.org/2020/1052.pdf (more information here).

Recommendation

We recommend clearly distinguishing messages that must be sent privately to other peers (through the authenticated channel), and messages which must be broadcast publicly to all peers (and ensuring robustness either through a trusted relay or through an echo mechanism).

Status Details

This has been addressed in PR #205 by introducing an extra echo round for messages that are meant to be broadcast.

4.3 ZKP Proofs sent before collecting all responses

Finding ID: KS-AMC-F-06

Severity: Medium

Status: Remediated

Location: `ecdsa/cggmp/sign/round_1.go`

`ecdsa/cggmp/signSix/round_1.go`

Description and Impact Summary

At the second step of round 2 of the presign phase, both in the 3-round and the 6-rounds variants (Fig. 7 and 9 of the paper, respectively), the two `psi` proofs should be computed and sent only when the `enc-elg` proof has been verified for all parties.

Round 2.

1. Upon receiving $(ssid, j, K_j, G_j, \psi^0_{i,j})$ from $\mathcal{P}_j$, do:
   - Verify $\mathcal{M}(\mathbf{vrfy}, \Pi^{\mathsf{enc}}_i, (ssid, j), (\mathcal{I}_\varepsilon, K_j), \psi_{i,j}) = 1$.
2. When passing above verification for all $\mathcal{P}_j$, set $\Gamma_i = g^{\gamma_i}$ and do:

   For every $j \neq i$, sample $r_{i,j}, s_{i,j}, \hat{r}_{i,j}, \hat{s}_{i,j} \leftarrow \mathbb{Z}_{N_j}$, $\beta_{i,j}, \hat{\beta}_{i,j} \leftarrow \mathcal{J}$ and compute:
   - $D_{j,i} = (\gamma_i \odot K_j) \oplus \mathsf{enc}_j(-\beta_{i,j}\,,\, s_{i,j})$ and $F_{j,i} = \mathsf{enc}_i(\beta_{i,j}\,,\, r_{i,j})$.
   - $\hat{D}_{j,i} = (x_i \odot K_j) \oplus \mathsf{enc}_j(-\hat{\beta}_{i,j}\,,\, \hat{s}_{i,j})$ and $\hat{F}_{j,i} = \mathsf{enc}_i(\hat{\beta}_{i,j}\,,\, \hat{r}_{i,j})$.
   - $\psi_{j,i} = \mathcal{M}(\mathbf{prove}, \Pi^{\mathsf{aff\text{-}g}}_j, (ssid, i), (\mathcal{I}_\varepsilon, \mathcal{J}_\varepsilon, D_{j,i}, K_j, F_{j,i}, \Gamma_i); (\gamma_i, \beta_{i,j}, s_{i,j}, r_{i,j}))$.
   - $\hat{\psi}_{j,i} = \mathcal{M}(\mathbf{prove}, \Pi^{\mathsf{aff\text{-}g}}_j, (ssid, i), (\mathcal{I}_\varepsilon, \mathcal{J}_\varepsilon, \hat{D}_{j,i}, K_j, \hat{F}_{j,i}, X_i); (x_i, \hat{\beta}_{i,j}, \hat{s}_{i,j}, \hat{r}_{i,j}))$.
   - $\psi'_{j,i} = \mathcal{M}(\mathbf{prove}, \Pi^{\mathsf{log*}}_j, (ssid, i), (\mathcal{I}_\varepsilon, G_i, \Gamma_i, g); (\gamma_i, \nu_i))$.

   Send $(ssid, i, \Gamma_i, D_{j,i}, F_{j,i}, \hat{D}_{j,i}, \hat{F}_{j,i}, \psi_{j,i}, \hat{\psi}_{j,i}, \psi'_{j,i})$ to each $\mathcal{P}_j$.

In the implementation, they are instead sent after passing their sender's verification only, without waiting for all others. This is actually done in `round_1.go` rather than `round_2.go`.

```
    // verify Proof_enc
    err = round1.Psi.Verify(parameter, p.own.ssidWithBk, round1.KCiphertext, n, ownPed)
    if err != nil {
        return err
    }
    negBeta, countDelta, r, s, D, F, phiProof, err :=
cggmp.MtaWithProofAff_g(p.own.ssidWithBk, peer.para, p.paillierKey, round1.KCiphertext,
p.gamma, Gamma)
    if err != nil {
        return err
    }
    // psihat share proof: M(prove,Πaff-
g,(sid,i),(Iε,Jε,D^j,i,Kj,F^j,i,Xi);(xi,β^i,j,s^i,j,r^i,j)).
    negBetahat, countSigma, rhat, shat, Dhat, Fhat, psihatProof, err :=
cggmp.MtaWithProofAff_g(p.own.ssidWithBk, peer.para, p.paillierKey, round1.KCiphertext,
p.bkMulShare, p.bkpartialPubKey)
    if err != nil {
        return err
    }
```

In addition to make the code difficult to parse, this might introduce the possibility of (ZKP counterpart of) rogue key attacks, see for example here. The idea is that an adversary could delay their response until they manage to see every other parties' response, and craft ad-hoc proofs adaptively on those responses to, e.g., pass a subsequent verification step without a proper witness. Even though we are not able to provide a specific attack in this case, we consider this a serious deviation from the original protocol, so the issue is marked as medium severity.

Recommendation

We recommend moving this part of the protocol into the appropriate `round_2.go` file, and waiting until all the verification steps are passed before sending the ZKP response.

Status Details

This has been addressed in PR #207.

4.4 Wrong modulo reduction

Finding ID: KS-AMC-F-07

Severity: Low

Status: Remediated

Location: `ecdsa/cggmp/signSix/round_5.go:171`
`Ecdsa/cggmp/signSix/round_6.go:168`

Description and Impact Summary

In the red-alert #1 algorithm in the paper (Fig. 11), the $\mu_{i,j}$ values are computed modulo N_i while in the implementation they are computed modulo `nsquare`.

```
    // build peersMsg
    peersMsg := make(map[string]*Err1PeerMsg, len(p.peers))
    for _, peer := range p.peers {
        muij := new(big.Int).Exp(nAddone, new(big.Int).Neg(peer.round2Data.alpha), nsquare)
        muij.Mul(muij, peer.round2Data.d)
        muNthPower := new(big.Int).Mod(muij, nsquare)
        mu := muij.Exp(muNthPower, nthRoot, nsquare)
        muNPower := muNthPower
```

The same happens for red-alert #2 algorithm (Fig. 12) in `round_6.go`.

Even if the proof `psiMuProof` later on is reduced modulo N correctly, having the mu proofs belonging to a larger group might leak undesired information. At a very minimum invalidates the security proof of the scheme.

Recommendation

We recommend performing the correct modulo reduction.

Status Details

This has been addressed in PR #204. However, we notice that the correct reduction modulo N is done in two steps: first compute the *mu* proof modulo N^2, and then subsequently reduce it further modulo N:

```
174                     mu := muij.Exp(muNthPower, nthRoot, nsquare)
175   +                 mu.Mod(mu, n)
176   +                 psiMuProof, err :=
      paillierzkproof.NewNthRoot(paillierzkproof.NewS256(), p.own.ssidWithBk, mu,
      muNthPower, n)
```

We think it would be better to compute mu as an exponential modulo N directly. However, the proposed approach also works.

5. OTHER OBSERVATIONS

This section contains additional observations that are not directly related to the security of the code, and as such have no severity rating or remediation status summary. These observations are either minor remarks regarding good practice or design choices or related to implementation and performance. These items do not need to be remediated for what concerns security, but where applicable we include recommendations.

5.1 Protocol steps interleaved within different files

Observation ID: KS-AMC-O-01

Location: `various`

Description and Impact Summary

We observe that in the CGGMP implementation, the various files `round_X.go` do not completely reflect the corresponding steps as described in the paper. For example, `round_4.go` contains both proof generation from the paper's round 4 and verification of the same proofs which appear in the paper's round 5. This makes the flow of the program hard to parse to the paper.

Recommendation

We recommend using filenames that easily map to the paper's described rounds, for ease of reading.

5.2 Use of notation from outdated versions of paper

Observation ID: KS-AMC-O-02

Location: `various`

Description and Impact Summary

We notice that the code seems to refer to variables names using notation that does not match the most recent versions of the FROST paper (example: `Elli` rather than `rhoell` and `zi` rather than `si`).

Recommendation

We recommend using variable names that closely reflect those used in the paper for ease of reading.

Notes

This has been addressed in PR #189.

5.3 Limited testing for key refresh

Observation ID: KS-AMC-O-03

Location: `various`

Description and Impact Summary

We notice that the testing functions for the refresh protocol are very basic. For example, only one case with all shares having the same hierarchical level (0) is tested, basically reducing Birckhoff interpolation to Lagrange.

Recommendation

We recommend extending the test units to broader scenarios.

Notes

Additional tests have been introduced in PR #203.

APPENDIX A: ABOUT KUDELSKI SECURITY

Kudelski Security is an innovative, independent Swiss provider of tailored cyber and media security solutions to enterprises and public sector institutions. Our team of security experts delivers end-to-end consulting, technology, managed services, and threat intelligence to help organizations build and run successful security programs. Our global reach and cyber solutions focus is reinforced by key international partnerships.

Kudelski Security is a division of Kudelski Group. For more information, please visit https://www.kudelskisecurity.com.

Kudelski Security

Route de Genève, 22-24

1033 Cheseaux-sur-Lausanne

Switzerland

Kudelski Security

5090 North 40th Street

Suite 450

Phoenix, Arizona 85018

APPENDIX B: DOCUMENT HISTORY

VERSION	STATUS	DATE	AUTHOR	COMMENTS
0.1	Draft	21 September 2022	Tommaso Gagliardoni	First draft
0.2	Draft	22 September 2022	Tommaso Gagliardoni	Corrected typos, added code snippet
1.0	Proposal	14 October 2022	Tommaso Gagliardoni	Status updated according to patched code
1.1	Proposal	22 October 2022	Tommaso Gagliardoni	Status updated for KS-AMC-O-03
1.2	Final	3 November 2022	Tommaso Gagliardoni	Clarified commit number of audited repo

REVIEWER	POSITION	DATE	VERSION	COMMENTS
Nathan Hamiel	Head of Security Research	21 September 2022	0.1	
Nathan Hamiel	Head of Security Research	22 September 2022	0.2	
Tommaso Gagliardoni	Senior Cryptography Expert	14 October 2022	1.0	
Tommaso Gagliardoni	Senior Cryptography Expert	22 October 2022	1.1	
Tommaso Gagliardoni	Senior Cryptography Expert	3 November 2022	1.2	For public release

APPROVER	POSITION	DATE	VERSION	COMMENTS
Nathan Hamiel	Head of Security Research	21 September 2022	0.1	
Nathan Hamiel	Head of Security Research	22 September 2022	0.2	

APPROVER	POSITION	DATE	VERSION	COMMENTS
Tommaso Gagliardoni	Senior Cryptography Expert	14 October 2022	1.0	
Tommaso Gagliardoni	Senior Cryptography Expert	22 October 2022	1.1	
Tommaso Gagliardoni	Senior Cryptography Expert	3 November 2022	1.2	

APPENDIX C: SEVERITY RATING DEFINITIONS

Kudelski Security uses a custom approach when determining criticality of identified issues. This is meant to be simple and fast, providing customers with a quick at a glance view of the risk an issue poses to the system. As with anything risk related, these findings are situational. We consider multiple factors when assigning a severity level to an identified vulnerability. A few of these include:

- Impact of exploitation
- Ease of exploitation
- Likelihood of attack
- Exposure of attack surface
- Number of instances of identified vulnerability
- Availability of tools and exploits

SEVERITY	DEFINITION
High	The identified issue may be directly exploitable causing an immediate negative impact on the users, data, and availability of the system for multiple users.
Medium	The identified issue is not directly exploitable but combined with other vulnerabilities may allow for exploitation of the system or exploitation may affect singular users. These findings may also increase in severity in the future as techniques evolve.
Low	The identified issue is not directly exploitable but raises the attack surface of the system. This may be through leaking information that an attacker can use to increase the accuracy of their attacks.
Informational	Informational findings are best practice steps that can be used to harden the application and improve processes.