NEAR One PedPop+

Security Assessment

April 15, 2025

Prepared for:
Bowen Wang
Near One

Prepared by: **Joe Doyle and Marc Ilunga**

Table of Contents

Project Summary

Contact Information

The following project manager was associated with this project:

Sam Greenup, Project Manager
sam.greenup@trailofbits.com

The following engineering director was associated with this project:

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

The following consultants were associated with this project:

Joe Doyle, Consultant
joseph.doyle@trailofbits.com

Marc Ilunga, Consultant
marc.ilunga@trailofbits.com

Project Timeline

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

Date	Event
April 03, 2025	Pre-project kickoff call
April 09, 2025	Status update meeting #1
April 15, 2025	Delivery of report draft & Report readout meeting
TBD	Delivery of final comprehensive report

Executive Summary

Engagement Overview

NEAR One engaged Trail of Bits to review the security of the latest upgrade to its chain signature system. The chain signature system enables distributed ECDSA signing. The latest upgrade brings EdDSA support via the FROST protocol, a generic distributed key generation protocol (DKG) named PedPod+, and a new contract coordination protocol for key generation and resharing.

A team of 2 consultants conducted the review from March 31 to April 11, 2025, for 4 engineer-weeks of effort. Our testing efforts focused on ensuring that the PedPod+ DKG is implemented securely, that chain signatures use FROST securely, and that the coordination protocol is sound. With full access to source code and documentation, we performed static and dynamic testing of the codebase using automated and manual processes.

Observations and Impact

Overall, the codebase is well-structured and thoroughly documented. The coordination protocol for key generation and resharing specification is reasonably well documented. The implementation securely uses the Cait-Sith and FROST libraries.

However, we identified a medium security issue that could undermine the proactive security guarantees expected from frequent key resharing. This issue is caused by a lack of explicit erasure of old key material (TOB-NEARDKG-1).

We also identified several inconsistencies in the corruption assumptions made across different system components. In particular, current corruption assumptions for the broadcast are incompatible with those made for the signing protocols (TOB-NEARDKG-5). Moreover, the corruption model and expected security guarantees for changing thresholds are not explicitly documented (TOB-NEARDKG-4).

Recommendations

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

- **Remediate the findings disclosed in this report.** These findings should be addressed as part of a direct remediation or any refactor that may occur when addressing other recommendations.
- **Produce a security proof for PepPop+.** The novel generic DKG built by NEAR One aims to achieve stronger security, namely “Simulatability with aborts”. Although we did not identify obvious issues in the protocol, our time-bound review does not constitute formal proof of security. We recommend that NEAR One establish proof of security for its DKG to gain additional confidence in PedPop+.

- **Clarify and unify corruption assumptions.** After a threshold key is generated, it may be involved in both signing and resharing, and can also have its participant list and signing threshold changed. The security of each such action requires that an attacker has only limited control over the participants, but those requirements are not the same across the system as designed. We recommend specifying the limitations on corrupt participants in each part of a key's lifetime – e.g., what an attacker with more than $N/3$ but less than t parties can do.
- **Unify the key generation and coordination specification.** The existing specification of the coordination protocol gives a reasonable understanding of the protocol at a high level. However, it lacks certain details (e.g., all information passed between states) that can be found across the code base. We recommend unifying the specifications and information in the code base in a comprehensive document to facilitate understanding of the protocols.

Finding Severities and Categories

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

EXPOSURE ANALYSIS

Severity	*Count*
High	0
Medium	1
Low	1
Informational	4
Undetermined	0

CATEGORY BREAKDOWN

Category	*Count*
Cryptography	3
Data Validation	1
Denial of Service	1
Patching	1

Project Goals

The engagement was scoped to provide a security assessment of Near One's mpc libraries, its usage of the FROST library, and its novel generic DKG. Specifically, we sought to answer the following non-exhaustive list of questions:

- Is the PedPop+ DKG implemented securely?
- Does the code use FROST correctly and securely?
- Is the contract-based coordination protocol for key generation and resharing sound?
- Is the resharing protocol robust against key deletion attacks?
- Do the protocols correctly protect against corruption of limited numbers of participants?

Project Targets

The engagement involved reviewing and testing the targets listed below.

Near-One/mpc

Repository	https://github.com/Near-One/mpc
Version	`e7777ee50f16837d899b5ce53c5709a1201b85bf`
Type	Rust
Platform	Multiple

Near-One/cait-sith

Repository	https://github.com/Near-One/cait-sith
Version	`5e0ce40a16dc3e0889277f66bb2a6400d6ef36a5`
Type	Rust
Platform	Multiple

Project Coverage

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

- **PedPop+ specification.** We reviewed the specifications for the novel DKG introduced by NEAR One. Since PedPop+ builds on the original PedPop DKG, we focused on ensuring no obvious vulnerabilities or weakening of the original DKG were introduced. We did not perform any formal analysis or proof of security.
- **DKG and broadcast implementations.** We reviewed the implementation of PedPop+ and the associated "Authenticated Double-Echo Broadcast" used by the protocol to ensure they are correctly and securely implemented. We paid special attention to input validation and avoiding issues with zero shares.
- **Key generation and resharing coordination**. We manually reviewed the smart contract's specification and management of the key-management state machine. We focused on the soundness of the state machine and input validation.
- **FROST library integration.** The FROST library is used to enable support for the threshold Schnorr signatures. We reviewed the use of the library, ensuring they are used safely to perform threshold signing.

Automated Testing

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

Test Harness Configuration

We used the following tools in the automated testing phase of this project:

Tool	Description	Policy
`cargo-audit`	A Cargo subcommand that can be used to audit projects dependencies for known vulnerabilities	Appendix C
`cargo-llvm-cov`	A Cargo plugin for generating LLVM source–based code coverage	Appendix C
Clippy	A Rust linter used to catch common mistakes and unidiomatic Rust code	Appendix C
Dylint	An open-source Rust linter developed by Trail of Bits to identify common code quality issues and mistakes in Rust code	Appendix C
Semgrep	An open-source static analysis tool for finding bugs and enforcing code standards when editing or committing code and during build time	Appendix C

Areas of Focus

Our automated testing and verification work focused on the following system properties:

- General code quality issues and unidiomatic code patterns
- Issues related to error handling
- Unit testing coverage
- General issues with dependency management and known vulnerable dependencies

Test Results

The results of this focused testing are detailed below.

Test Results

The results of this focused testing are detailed below.

`Cargo-audit`

The `cargo-audit` tool identified several outdated dependencies with advisories (TOB-NEARDKG-2)

`cargo-llvm-cov`

The coverage report shows satisfactory test coverage for the Cait-Sith codebase. However, the test coverage of the mpc codebase remains insufficient as reported in TOB-NEARONE-11.

Filename	Function Coverage	Line Coverage	Region Coverage
compat/mod.rs	100.00% (11/11)	96.49% (55/57)	87.50% (21/24)
crypto.rs	90.00% (9/10)	94.23% (49/52)	90.00% (9/10)
ecdsa/dkg_ecdsa.rs	100.00% (9/9)	98.22% (166/169)	79.41% (54/68)
ecdsa/math.rs	85.29% (29/34)	86.27% (132/153)	87.10% (54/62)
ecdsa/mod.rs	100.00% (2/2)	100.00% (9/9)	100.00% (2/2)
ecdsa/presign.rs	60.00% (6/10)	87.15% (217/249)	76.19% (64/84)
ecdsa/sign.rs	73.33% (11/15)	78.31% (213/272)	69.79% (67/96)
ecdsa/test.rs	100.00% (16/16)	98.51% (199/202)	90.00% (90/100)
ecdsa/triples/batch_random_ot.rs	100.00% (28/28)	98.26% (339/345)	95.12% (156/164)
ecdsa/triples/bits.rs	95.12% (39/41)	95.95% (213/222)	97.76% (131/134)
ecdsa/triples/correlated_ot_extension.rs	100.00% (7/7)	93.90% (77/82)	86.67% (26/30)
ecdsa/triples/generation.rs	84.62% (11/13)	90.53% (937/1035)	82.09% (275/335)
ecdsa/triples/mod.rs	50.00% (1/2)	47.89% (34/71)	38.46% (5/13)
ecdsa/triples/mta.rs	100.00% (16/16)	97.55% (159/163)	89.33% (67/75)
ecdsa/triples/multiplication.rs	100.00% (26/26)	100.00% (289/289)	90.91% (120/132)
ecdsa/triples/random_ot_extension.rs	100.00% (13/13)	97.83% (180/184)	91.25% (73/80)
echo_broadcast.rs	100.00% (32/32)	93.56% (421/450)	88.04% (184/209)
eddsa/dkg_ed25519.rs	100.00% (9/9)	98.22% (166/169)	79.41% (54/68)
eddsa/mod.rs	100.00% (2/2)	100.00% (9/9)	100.00% (2/2)
eddsa/sign.rs	86.36% (19/22)	93.98% (437/465)	79.62% (125/157)
eddsa/test.rs	100.00% (10/10)	98.20% (164/167)	86.90% (73/84)
generic_dkg.rs	78.38% (29/37)	83.36% (491/589)	75.31% (183/243)
participants.rs	95.65% (22/23)	94.74% (144/152)	91.03% (71/78)
proofs/dlog.rs	100.00% (5/5)	100.00% (65/65)	100.00% (6/6)
proofs/dlogeq.rs	100.00% (5/5)	100.00% (76/76)	100.00% (6/6)
protocol/internal.rs	98.25% (56/57)	98.10% (310/316)	94.20% (130/138)
protocol/mod.rs	66.67% (8/12)	66.67% (84/126)	68.42% (52/76)
serde.rs	100.00% (10/10)	100.00% (54/54)	90.48% (19/21)
Totals	**92.45% (441/477)**	**91.88% (5689/6192)**	**84.86% (2119/2497)**

Figure 1: Cait-Sith library testing coverage

Clippy and Dylint

Running Clippy in pedantic mode and Dylint identifies several unidiomatic patterns and potential issues related to casting that should be investigated. In appendix C, we describe how to run Clippy in pedantic mode and efficiently triage these results using the SARIF file format.

Semgrep

We ran Semgrep Pro on the codebase using both the default configuration and our internal rulesets. This analysis did not identify any security issues in the codebase.

Summary of Findings

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

ID	Title	Type	Severity
1	Generic DKG does not delete temporary randomness	Cryptography	**Medium**
2	Outdated dependencies with advisories	Patching	**Informational**
3	DKG assertions are left to the caller	Data Validation	**Informational**
4	Unclear security model when resharing with a different threshold	Cryptography	**Informational**
5	Broadcast corruption threshold may not match signing threshold	Denial of Service	**Low**
6	Reshare public key checks can occur earlier	Cryptography	**Informational**

Detailed Findings

1. Generic DKG does not delete temporary randomness

Severity: **Medium**	Difficulty: **High**
Type: Cryptography	Finding ID: TOB-NEARDKG-1
Target: `cait-sith/src/generic_dkg.rs`	

Description

The generic DKG allows participants to create a shared signing key or update shares associated with an existing one. The DKG protocol's state contains several secret values that should be discarded after completion. The implementation does not discard these values, negatively impacting the scheme's proactive security.

The resharing protocol periodically refreshes participants' shares to enhance security. Refreshing shares limit the effect of compromises over time since the old shares are independent of the new ones, though they correspond to the same signing key. In other words, each refresh operation defines an epoch, and the compromise of a t - 1 participant in two adjacent epochs should not leak any information about the signing key. Deleting old shares after each resharing is crucial; retaining them allows attackers to compromise participants' states at different times, accumulating enough shares to recover the secret. Failure to delete old shares lets an attacker with t-1 compromised states in the current epoch use old shares to get the remaining state and reconstruct the secret, negating resharing's proactive security benefits.

Figure 4.1 shows the running of the reshare protocol. The reshare protocol relies on the function `do_keyshare`, which runs the generic DKG protocol with the old signing share as input. However, nowhere in the code base is the old share deleted after a successful run of the resharing protocol.

```
/// reshares the keyshares between the parties and allows changing the threshold
pub(crate) async fn do_reshare<C: Ciphersuite>(
    chan: SharedChannel,
    participants: ParticipantList,
    me: Participant,
    threshold: usize,
    old_signing_key: Option<SigningShare<C>>,
    old_public_key: VerifyingKey<C>,
```

```
    old_participants: ParticipantList,
) -> Result<KeygenOutput<C>, ProtocolError> {
    // prepare the random number generator
    let rng = OsRng;

    let intersection = old_participants.intersection(&participants);
    // either extract the share and linearize it or set it to zero
    let secret = old_signing_key
        .map(|x_i| intersection.generic_lagrange::<C>(me) * x_i.to_scalar())
        .unwrap_or(<C::Group as Group>::Field::zero());

    let old_reshare_package = Some((old_public_key, old_participants));
    let keygen_output = do_keyshare::<C>(
        chan,
        participants,
        me,
        threshold,
        secret,
        old_reshare_package,
        rng,
    )
    .await?;

    Ok(keygen_output)
}
```

Figure 4.1: The resharing protocol based on the generic DK (`cait-sith/src/generic_dkg.rs#L634-L666`)

Exploit Scenario

The states of t - 1 mpc nodes are exposed due to an initial malware compromise. After getting rid of the malware, a refresh protocol is executed to render the previously compromised shares useless. The malware compromises another node and recovers undeleted shares from the previous epoch. The malware has, therefore, enough shares to reconstruct the secret key.

Recommendations

Short term, ensure all temporary secret values are deleted upon completing any DKG protocol.

Long term, implement the zeroize trait for all data types that need to be discarded. Zeroizing temporary secret value is generally a good practice. However, the Rust compiler does not guarantee that data has not been copied into other locations, so we recommend documenting this limitation and providing guidance to users to help protect in-memory values.

References

- A pitfall of Rust's move/copy/drop semantics and zeroing data

2. Outdated dependencies with advisories

Severity: **Informational**	Difficulty: **High**
Type: Patching	Finding ID: TOB-NEARDKG-2
Target: `cait-sith/, mpc/`	

Description

The cait-sith and mpc client codebases use outdated dependencies with security advisories. The table below shows the list of outdated dependencies, several of which are unmaintained. The dependencies do not pose a direct threat to the system. However, unmaintained dependencies may hide vulnerabilities that could endanger the system when discovered.

Outdated dependencies in Cait Sith

Dependency	Version	Advisory ID	Description
ansi_term	0.12.1	RUSTSEC-2021-0139	unmaintained
atty	0.2.14	RUSTSEC-2024-0375	unmaintained
instant	0.1.13	RUSTSEC-2024-0384	unmaintained
paste	1.0.15	RUSTSEC-2024-0436	unmaintained
proc-macro-error	1.0.4	RUSTSEC-2024-0370	unmaintained

Outdated dependencies in mpc

Dependency	Version	Advisory ID	Description

dotenv	0.15.0	RUSTSEC-2021-0141	unmaintained
instant	0.1.13	RUSTSEC-2024-0384	unmaintained
lock_api	0.3.4	RUSTSEC-2020-0070	Data races
memmap	0.7.0	RUSTSEC-2020-0077	unmaintained
parity-wasm	0.41.0	RUSTSEC-2024-0370	unmaintained
paste	1.0.15	RUSTSEC-2024-0436	unmaintained
proc-macro-error	1.0.4	RUSTSEC-2024-0370	unmaintained
protobuf	2.28.0	RUSTSEC-2024-0437	Crash due to uncontrolled recursion
ring	0.16.20	RUSTSEC-2025-0009	Some AES functions may panic when overflow checking is enabled
wee_alloc	0.4.5	RUSTSEC-2022-0054	unmaintained

Recommendations

Short term, update the outdated dependencies to ensure the code is not affected by any potential vulnerability.

Long term, run `cargo-audit` as part of the CI/CD pipeline and ensure that the developers are alerted to any vulnerable dependencies that are detected.

3. DKG assertions are left to the caller

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-NEARDKG-3
Target: `cait-sith/src/eddsa/dkg_ed25519.rs`	

Description

The key generation and reshare protocols depend on invariants to ensure they run correctly and securely. These invariants are implemented in specific functions outside of the DKG core and are not systematically enforced by the DKG core. Therefore, the caller is effectively responsible for enforcing the invariants.

Figure 3.1 shows the key generation function for FROST. The implementation enforces the key generations invariant by calling `assert_keygen_invariants`. It would have been possible to forget the call to `assert_keygen_invariants` and still run the protocol successfully.

```
pub fn keygen(
    participants: &[Participant],
    me: Participant,
    threshold: usize,
) -> Result<impl Protocol<Output = KeygenOutput>, InitializationError> {
    let ctx = Context::new();
    let participants = assert_keygen_invariants(participants, me, threshold)?;
    let fut =
        do_keygen(ctx.shared_channel(), participants, me, threshold).map(|x|
x.map(Into::into));
    Ok(make_protocol(ctx, fut))
}
```

Figure 3.1: FROST Key gen (`cait-sith/src/eddsa/dkg_ed25519.rs#L13-L23`)

Similarly, the reshare protocol enforces the reshare invariants by calling `reshare_assertions`; however, this line could have been forgotten while not preventing to run of the core DKG.

```
/// Performs the Ed25519 Reshare protocol
pub fn reshare(
    old_participants: &[Participant],
    old_threshold: usize,
    old_signing_key: Option<SigningShare>,
    old_public_key: VerifyingKey,
    new_participants: &[Participant],
```

```
    new_threshold: usize,
    me: Participant,
) -> Result<impl Protocol<Output = KeygenOutput>, InitializationError> {
    let ctx = Context::new();
    let threshold = new_threshold;
    let (participants, old_participants) = reshare_assertions::<E>(
        new_participants,
        me,
        threshold,
        old_signing_key,
        old_threshold,
        old_participants,
    )?;
    let fut = do_reshare(
        ctx.shared_channel(),
        participants,
        me,
        threshold,
        old_signing_key,
        old_public_key,
        old_participants,
    )
    .map(|x| x.map(Into::into));
    Ok(make_protocol(ctx, fut))
}
```

Figure 3.2: FROST reshare (`cait-sith//src/eddsa/dkg_ed25519.rs#L25-L56`)

Recommendations

Short term, enforce the DKG assertions in the DKG core.

Long term, ensure that assertion and invariants are systematically enforced.

4. Unclear security model when resharing with a different threshold

Severity: **Informational**	Difficulty: **Not Applicable**
Type: Cryptography	Finding ID: TOB-NEARDKG-4
Target: `cait-sith/src/generic_dkg.rs`	

Description

Threshold signing and distributed key generation algorithms are parameterized by a threshold parameter t, which indicates the number of parties that are assumed to be acting honestly. However, the resharing version of the PedPop+ DKG algorithm is parameterized by a previous threshold, old_t, and a new threshold t. The precise security requirements of this algorithm have not been specified, and when $t \neq old\_t$, which participants an attacker can control without compromising the protocol is unclear. Although the core algorithm is identical to generating a fresh key with a threshold of t, there are two main cases with potential security implications:

- If $old\_t < t$, then an adversary who is able to compromise old_t previous shares can reconstruct the secret without needing to corrupt t parties. If previous shares are not deleted properly, or if the adversary is able to derive information about previous shares from other ephemeral data, an adversary could reconstruct enough shares even after the resharing protocol concludes.
- If $old\_t > t$, then an adversary who compromised t parties would become able to compromise the key after a resharing event.

There does not appear to be any direct attack possible if the adversary controls less than old_t participants before the reshare, less than t participants after, and less than both old_t and t participants during the reshare. However, these security assumptions have not been explicitly specified and require further analysis.

Recommendations

Short term, define and document the security model of the key resharing process, and what participants an adversary is allowed to control during different phases of a long-lived threshold signing key.

Long term, develop an explicit security proof to ensure that the PedPop+ key generation protocol is secure both in key generation and key resharing modes.

5. Broadcast corruption threshold may not match signing threshold

Severity: **Low**	Difficulty: **High**
Type: Denial of Service	Finding ID: TOB-NEARDKG-5
Target: `cait-sith/src/{generic_dkg.rs,echo_broadcast.rs}`	

Description

The PedPop+ protocol for key generation and resharing relies on a reliable broadcast primitive, which is implemented through an echo-broadcast protocol. The echo-broadcast protocol's correctness assumes that $N > 3f$, where N is the total number of participants and f is the maximum number of Byzantine participants. In the DKG setting, if an adversary is able to control $t - 1$ participants, and $f < t - 1$, it would be possible for that adversary to perform a split-view attack, where different groups of honest participants receive different results from each broadcast. This attack would allow corrupt participants to send different polynomials to different parties. In this case, the DKG protocol could succeed while the derived shares correspond to different polynomials. If participants do not have a way to recover a key from a previous session, the group could lose access to the key.

The parameter f in the echo-broadcast is always set to $(N - 1)/3$, as shown below in figure 5.1, which can easily be below $t - 1$.

```
/// Outputs the necessary Echo-Broadcast thresholds based on the
/// total number of participants. The  threshold in echo-broadcast
/// should be exceeded to continue with the next phase (it is not
/// sufficient to only hit the threshold but this should be exceeded).
/// The threshold are taken from the book:
/// "Introduction to Reliable and Secure Distributed Programming,
/// by C. Cachin, R. Guerraoui, and L. Rodrigues"
fn echo_ready_thresholds(n: usize) -> (usize, usize) {
    // case where no malicious parties are assumed: when n <= 3/
    // In this case the echo and ready thresholds are both 0
    // later we compare if we have collected more votes than these thresholds
    if n <= 3 {
        return (0, 0);
    }
    // we should always have n >= 3*threshold + 1
    let broadcast_threshold = (n - 1) / 3;
    let echo_threshold = (n + broadcast_threshold) / 2;
    (echo_threshold, broadcast_threshold)
}
```

Figure 5.1: The broadcast threshold is computed based only upon the number of participants (`cait-sith/src/echo_broadcast.rs#60–78`)

In fact, since the smart contract requires that the threshold is at least 60% of the total participants, as shown below in figure 5.2, the case where $t - 1 > f$ will always be the case.

```
/// Ensures that the threshold `k` is sensible and meets the absolute and minimum requirements.
/// That is:
/// - threshold must be at least `MIN_THRESHOLD_ABSOLUTE`
/// - threshold can not exceed the number of shares `n_shares`.
/// - threshold must be at least 60% of the number of shares (rounded upwards).
pub fn validate_threshold(n_shares: u64, k: Threshold) -> Result<(), Error> {
    if k.value() > n_shares {
        return Err(InvalidThreshold::MaxRequirementFailed
            .message(format!("cannot exceed {}, found {:?}", n_shares, k)));
    }
    if k.value() < MIN_THRESHOLD_ABSOLUTE {
        return Err(InvalidThreshold::MinAbsRequirementFailed.into());
    }
    let percentage_bound = (3 * n_shares + 4) / 5; // minimum 60%
    if k.value() < percentage_bound {
        return Err(InvalidThreshold::MinRelRequirementFailed.message(format!(
            "require at least {}, found {:?}",
            percentage_bound, k
        )));
    }
    Ok(())
}
```

Figure 6.2: The threshold configured in the smart contract is forced to be at least 60% (mpc/libs/chain-signatures/contract/src/primitives/thresholds.rs#52–73)

This attack cannot corrupt the key, and if there is a key backup system, it is only able to cause a temporary denial of service.

Exploit Scenario

Alice gains access to more than ⅓ of the share-holders in a key. During the next reshare, she performs a split-view attack and causes the honest participants to generate shares from two different polynomials, making that newly reshared key unusable. The key must then be reconstructed from backups, causing a denial of service.

Recommendations

Short term, document and specify the attacker model for key generation, especially focused on what malicious behaviors should cause a reshare to fail. Ensure that the overall system can back up and recover a key which has been corrupted. Consider mitigating this attack by adding an additional check to ensure that all participants have received the same commitments – for example, by computing and sending a hash of the sum of the committed polynomials when sending evaluations in round 3.

Long term, ensure that all protocols behave correctly up to their maximum corruption thresholds.

6. Reshare public key checks can occur earlier

Severity: **Informational**	Difficulty: **Not Applicable**
Type: Cryptography	Finding ID: TOB-NEARDKG-6
Target: `cait-sith/src/generic_dkg.rs`	

Description

When running the reshare variant of the DKG, a final check is performed to ensure that the new key shares correspond to the same public key. However, this check is only performed after an additional round of communication, as shown below in figure 6.1.

```
let commitments_and_proofs_map = do_broadcast(
    &mut chan,
    &participants,
    &me,
    (commitment, proof_of_knowledge),
)
.await?;

// Start Round 3
let wait_round_3 = chan.next_waitpoint();
for p in participants.others(me) {
    let (commitment_i, proof_i) = commitments_and_proofs_map.index(p);
…
    // send the evaluation privately to participant p
    chan.send_private(wait_round_3, p, &signing_share_to_p)
        .await;
}
…
// Start Round 4
// receive evaluations from all participants
let mut seen = ParticipantCounter::new(&participants);
seen.put(me);
while !seen.full() {
    let (from, signing_share_from): (Participant, SigningShare<C>) =
        chan.recv(wait_round_3).await?;
    if !seen.put(from) {
        continue;
    }
…
}

// cannot fail as all_commitments at least contains my commitment
let all_commitments_vec = all_full_commitments.into_vec_or_none().unwrap();
let all_commitments_refs = all_commitments_vec.iter().collect();

let verifying_key = public_key_from_commitments(all_commitments_refs)?;

// In the case of Resharing, check if the old public key is the same as the new one
```

```
if let Some(old_vk) = old_verification_key {
    // check the equality between the old key and the new key without failing the unwrap
    if old_vk != verifying_key {
        return Err(ProtocolError::AssertionFailed(
            "new public key does not match old public key".to_string(),
        ));
    }
};
```

Figure 6.1: The polynomial commitments received in the broadcast are not validated until after a round of sending and receiving (`cait-sith/src/generic_dkg.rs#463–561`)

If an attacker attempts to modify the public key by submitting an incorrect value for their public key share, honest participants will generate and send evaluations before failing.

Since the generated polynomials all have degree at least $t - 1$, these evaluations do not reveal any secret information, but the additional communication is not necessary and can be avoided by performing the public key check as soon as the commitments to polynomials have been received.

Recommendations

Short term, consider modifying the key generation protocol to perform the public key check immediately after receiving all participants' committed polynomials.

Long term, validate received data as early as possible to ensure that no further actions are taken if any participant detects misbehavior.

A. Vulnerability Categories

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

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

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

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

B. Code Quality Findings

- **Functions with too many lines.** The function `reliable_broadcast_receive_all` has a long body. Functions with many lines hinder the readability of the codebase and may also complicate unit testing.
- **Unnamed domain separators.** The `generic_dkg.rs` code uses a domain separator to instantiate several independent hash functions. This domain separator is implemented as a counter that is incremented for each instantiation, which is hard to track and potentially error-prone. Using an enum might be cleaner in this instance.

```
let mut domain_separator = 0;
// Make sure you do not call do_keyshare with zero as secret on an old participant
let (old_verification_key, old_participants) =
    assert_keyshare_inputs(me, &secret, old_reshare_package)?;

// Start Round 0
let mut my_session_id = [0u8; 32]; // 256 bits
OsRng.fill_bytes(&mut my_session_id);
let session_ids = do_broadcast(&mut chan, &participants, &me, my_session_id).await?;

// Start Round 1
// generate your secret polynomial p with the constant term set to the secret
// and the rest of the coefficients are picked at random
// because the library does not allow serializing the zero and identity term,
// this function does not add the zero coefficient
let session_id = domain_separate_hash(domain_separator, &session_ids);
domain_separator += 1;
let secret_coefficients = generate_secret_polynomial::<C>(secret, threshold, &mut
rng);
```

Figure B.1: Hash domain separator implemented as an increasing counter (cait-sith/src/generic_dkg.rs#408–425)

- **Erroneous error message.** An error is returned during resharing if an old participant runs the resharing protocol with a zero share. However, the error message is unrelated to what is happening.

```
if !old_participants.contains(me) {
    return Err(ProtocolError::AssertionFailed(
        format!("{me:?} is running Resharing with a zero share but does belong to
the old participant set")));
    }
```

Figure B.2:Erroneous error message (cait-sith/src/generic_dkg.rs#48–51)

C. Automated Testing

This section describes the setup of the automated analysis tools used during this audit.

cargo-audit

The `cargo-audit` Cargo plugin identifies known vulnerable dependencies in Rust projects. It can be installed using `cargo install cargo-audit`. To run the tool, run `cargo audit` in the crate root directory.

cargo-llvm-cov

The `cargo-llvm-cov` Cargo plugin is used to generate LLVM source–based code coverage data. The plugin can be installed via the command `cargo install cargo-llvm-cov`. To run the tool, run the command `cargo llvm-cov` in the crate root directory.

Clippy

The Rust linter Clippy can be installed using `rustup` by running the command `rustup component add clippy`. Invoking `cargo clippy -- -W clippy::pedantic` in the root directory of the project runs Clippy in pedantic mode.

Dylint

Dylint is a linter for Rust developed by Trail of Bits. It can be installed by running the command `cargo install cargo-dylint dylint-link`. To run Dylint, we added a `Cargo.toml` file to the root of the repository with the following content.

```
[workspace.metadata.dylint]
libraries = [
  { git = "https://github.com/trailofbits/dylint", pattern = "examples/general/*" },
]
```

Figure C.1: Metadata required to run Dylint

To run the tool, run `cargo dylint --all --workspace`.

Semgrep

Semgrep can be installed using `pip` by running `python3 -m pip install semgrep`. To run Semgrep on a codebase, run `semgrep --config "<CONFIGURATION>"` in the root directory of the project. Here, `<CONFIGURATION>` can be a single rule, a directory of rules, or the name of a ruleset hosted on the Semgrep registry. Trail of Bits' public ruleset can be used by running `semgrep --config "p/trailofbits"`.

D. Pedpop+ Specification Findings

- The resharing scheme specification does not include the computation of the polynomial evaluations.

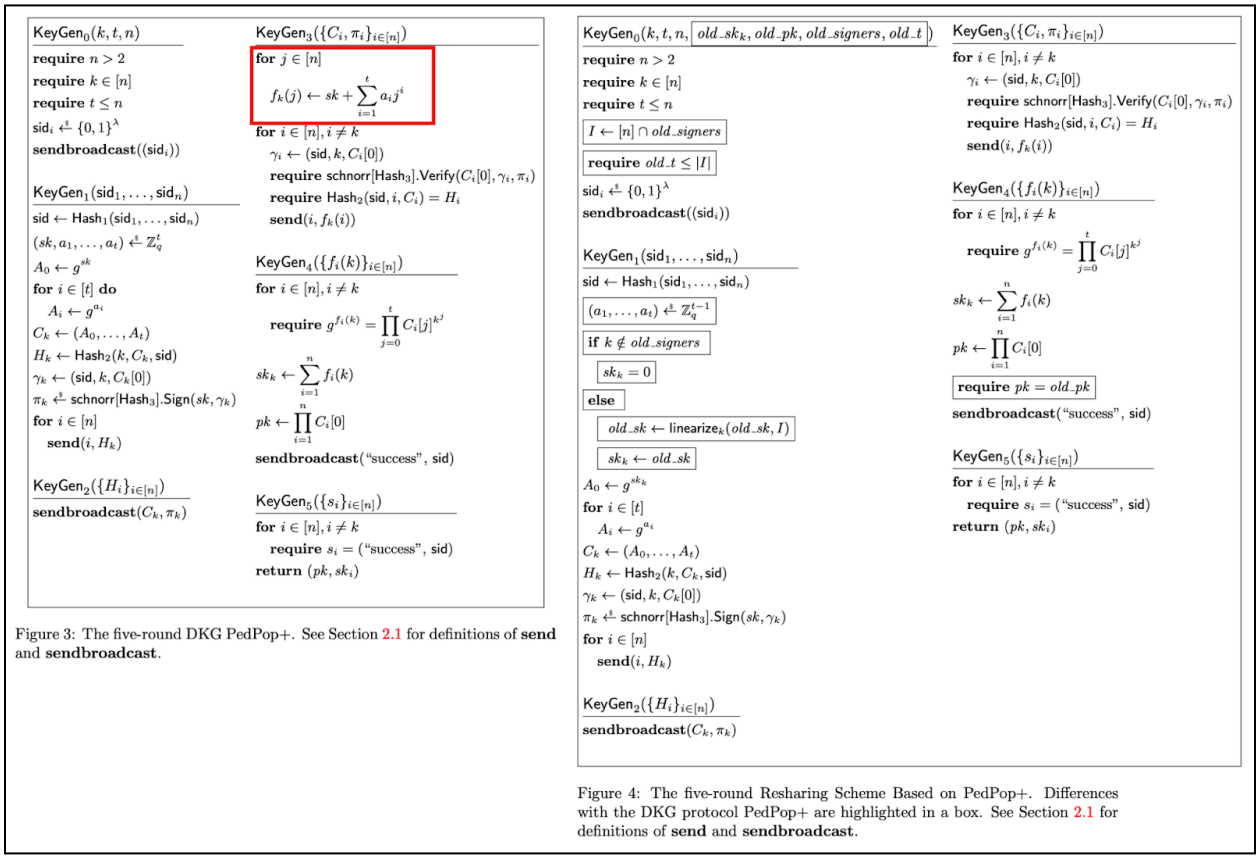

Figure 3: The five-round DKG PedPop+. See Section 2.1 for definitions of **send** and **sendbroadcast**.

Figure 4: The five-round Resharing Scheme Based on PedPop+. Differences with the DKG protocol PedPop+ are highlighted in a box. See Section 2.1 for definitions of **send** and **sendbroadcast**.

Figure D.1: The DKG and resharing specifications, with the missing computation highlighted.

- Figure 2 of the PedPop+ specification shows the original PedPop protocol. The final key share output is indexed by `i` rather than k.

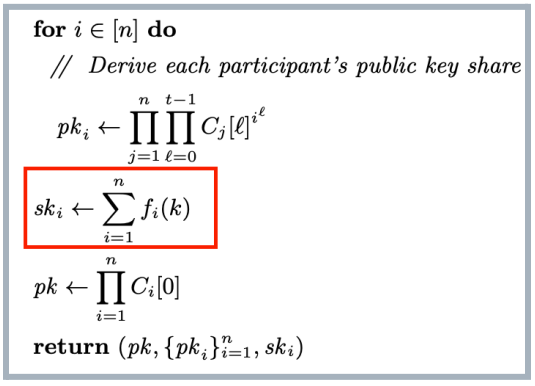

Figure D.2: Typo in the KeyGen_3 algorithm

About Trail of Bits

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

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

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

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries and government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

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

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

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

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

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

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