NEAR One Confidential Key Derivation

Security Assessment

December 23, 2025

Prepared for:
Mårten Blankfors
NEAR One

Prepared by: **Filipe Casal and Scott Arciszewski**

Table of Contents

Project Summary

Contact Information

The following project manager was associated with this project:

Kimberly Espinoza, Project Manager
kimberly.espinoza@trailofbits.com

The following engineering director was associated with this project:

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

The following consultants were associated with this project:

Filipe Casal, Consultant
filipe.casal@trailofbits.com

Scott Arciszewski, Consultant
scott.arciszewski@trailofbits.com

Project Timeline

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

Date	Event
September 18, 2025	Pre-project kickoff call
September 29, 2025	Status update meeting #1
October 6, 2025	Delivery of report draft and report readout meeting
October 15, 2025	Delivery of final comprehensive report
December 19, 2025	Completion of fix review
December 23, 2025	Delivery of updated report with fix review appendix

Executive Summary

Engagement Overview

NEAR One engaged Trail of Bits to review the security of a Confidential Key Derivation (CKD) protocol and implementation.

The CKD protocol aims to enable an application running within a trusted execution environment (TEE) to derive a deterministic and confidential key using a set of NEAR smart contracts and nodes participating in a multi-party computation (MPC) protocol.

A team of two consultants conducted the review from September 22 to October 3, 2025, for a total of four engineer-weeks of effort. Our testing efforts focused on verifying the security goals of the designed protocol, specifically the confidentiality of the derived key, as well as on validating the implementation against these goals. 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

The review identified two high-severity findings that compromise the core security objective of the CKD protocol: the confidentiality of the derived key. First, as a stand-alone protocol, CKD allows a malicious coordinator to control the derived key (TOB-NEARCKD-1). We validated this attack by formally modeling the protocol in ProVerif.

Second, there are no access controls in the smart contract that handles the CKD responses from the MPC nodes (TOB-NEARCKD-2), which means that any NEAR network account could perform the attack described in TOB-NEARCKD-1.

The review also identified several medium-severity findings related to input validation and error handling. Two findings (TOB-NEARCKD-3 and TOB-NEARCKD-7) allow malicious participants to cause runtime panics in other protocol participants by providing malformed data. We also identified an edge case in the resharing protocol (TOB-NEARCKD-8) that will always fail.

Additionally, the review uncovered multiple low- and informational-severity findings related to edge case handling, serialization issues, timeout handling, and supply chain security in the CI/CD pipeline.

Recommendations

Based on the findings identified during the security review, Trail of Bits recommends that NEAR One take the following steps before deploying the CKD protocol:

- **Remediate the findings disclosed in this report.** These findings should be addressed through direct fixes or broader refactoring efforts.

- **Strengthen the CKD protocol.** Consider integrating zero-knowledge proofs of discrete-log equality to enable applications to verify the correctness of the derived key. This would prevent certain classes of attacks but still allow the MPC nodes to derive the CKD keys if they were to collude or if they were directly attacked (TOB-NEARCKD-15). Perform a comparative analysis of the requirements, security guarantees, and hardness assumptions between CKD and the considered alternatives. Consider adapting well-studied and published protocols in the verifiable threshold oblivious pseudorandom function (OPRF) literature, using Dstack's KMS in KMS mode to derive a secret key bound to each application.

- **Strengthen input validation throughout the codebase.** Implement comprehensive validation of all participant-provided data. Use checked arithmetic operations and validate bounds and keys before indexing into data structures. Add fuzzing harnesses and `kani` proofs to functions that interact with potentially malicious or malformed data.

- **Formally specify the protocol and its security properties.** Create a formal specification of the CKD protocol and the resharing and refresh protocols to explicitly document all security assumptions, invariants, and validation requirements. Annotate the implementation with references to specification steps to ensure completeness.

Finding Severities and Categories

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

EXPOSURE ANALYSIS

Severity	*Count*
High	2
Medium	3
Low	3
Informational	7
Undetermined	0

CATEGORY BREAKDOWN

Category	*Count*
Access Controls	1
Configuration	3
Cryptography	4
Data Validation	6
Denial of Service	1

Project Goals

The engagement was scoped to provide a security assessment of the NEAR One CKD protocol and implementation. Specifically, we sought to answer the following non-exhaustive list of questions:

- Does the proposed CKD protocol achieve its security objectives?
- Does the CKD implementation follow the proposed protocol?
- Is the implementation robust against malicious or malformed data?
- Do the NEAR smart contracts that bridge the application with the nodes participating in the MPC protocol introduce security issues in the implementation?
- Does the implementation follow Rust best practices?
- Are the repositories’ CI pipelines secure against supply chain attacks?

Project Targets

The engagement involved reviewing and testing the following targets.

threshold-signatures (confidential-key-derivation)

Repository	https://github.com/near/threshold-signatures
Version	`e67c26c6dc33498b7e0545df929caa293d1cfc4e`
Type	Rust
Platform	Native, TEE

mpc (request_app_private_key function)

Repository	https://github.com/near/mpc
Version	`2215b43ddf6887b238779534d5ae5b3638b94e83`
Type	Rust, NEAR contract
Platform	NEAR, Wasm

Project Coverage

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

- Manual code review of the `confidential_key_derivation` folder and the `request_app_private_key` smart-contract function, focusing on protocol correctness, the confidentiality of the key derived by the application, and denial-of-service and access controls issues
- Comparison of the protocol specification and its Rust implementation
- Formal modeling of the application view of the CKD protocol using ProVerif
- Manual code review of relevant components of `frost-core` in search of side-channel leakage
- Static analysis of the Rust code and repository GitHub actions using the tools detailed in the Automated Testing section
- Fuzz testing of the `Borsh` and `bincode` serialization-deserialization round-trip for the `AppId` type using a `test-fuzz` harness
- Exhaustive testing of the `Participant::scalar` function: tested all possible u32 inputs to guarantee that the `scalar` function never returns zero
- Manual code review of utilities in the `threshold-signatures` codebase: `polynomials`, `commitments`, and `participants`

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. The setup and usage options for all tools are described in appendix D.

Tool	Description
CodeQL	A code analysis engine developed by GitHub to automate security checks
Semgrep	An open-source static analysis tool for finding bugs and enforcing code standards when editing or committing code and during build time
`zizmor`	A static analysis tool for GitHub Actions
Clippy	An open-source Rust linter used to catch common mistakes and unidiomatic Rust code
Dylint	A tool for running Rust lints from dynamic libraries
`test-fuzz`	A tool to quickly set up a fuzzing campaign in Rust
`cargo-edit`	A tool for quickly identifying outdated crates
`cargo-audit`	An open-source tool for checking dependencies against the RustSec advisory database
`cargo-llvm-cov`	A Cargo plugin for generating LLVM source–based code coverage
`cargo-mutants`	An open-source mutation testing tool for Rust

Areas of Focus

Our automated testing and verification work focused on the following:

- General code quality issues and unidiomatic code patterns

- Data validation issues with fuzzing
- Security of CI and GitHub actions
- Dependencies that are outdated or vulnerable to known attacks
- Gaps in test coverage

Test Results

The results of this focused testing are detailed below.

CodeQL and Semgrep

CodeQL and Semgrep did not identify any relevant findings in the codebase.

zizmor

Running `zizmor` on the codebase identified TOB-NEARCKD-11, TOB-NEARCKD-12, and TOB-NEARCKD-14. We recommend including `zizmor` in the project's CI/CD pipeline.

Clippy

Clippy identified the truncating cast that leads to finding TOB-NEARCKD-4. Additionally, it identified several code quality issues reported in the Code Quality appendix. We recommend running Clippy with its standard ruleset together with the `or_fun_call` rule on every commit, and with its pedantic ruleset before every major release. If code patterns that are identified keep recurring in the codebase, add these rules to the regular Clippy CI runs.

Dylint

Dylint identified code quality issues reported in the Code Quality appendix. We recommend running Dylint's general and supplementary ruleset before every major release.

test-fuzz

We used `test-fuzz` to write a fuzzing harness that tests the serialization-deserialization round-trip for the `bincode` and `Borsh` flavors for the `AppId` type. This yielded no runtime errors during a one-hour-long fuzzing campaign. After adding the fuzzing harness and running `cargo test`, run `cargo test-fuzz fuzz_harness` to start the campaign.

```
use bincode::config;
use bincode::serde::{decode_from_slice, encode_to_vec};

#[test_fuzz::test_fuzz]
fn fuzz_harness(bytes: &[u8]) {
    let original = AppId::new(bytes);
    let mut buf = vec![];
    borsh::BorshSerialize::serialize(&original, &mut buf).unwrap();

    let decoded = AppId::deserialize_reader(&mut buf.as_slice()).unwrap();
    assert_eq!(decoded, original);
    assert_eq!(decoded.as_bytes(), &bytes[..]);

    let original = AppId::new(bytes);

    // Encode using bincode's binary format
    let encoded = encode_to_vec(&original, config::standard()).expect("bincode
encode");

    // Decode back into AppId
    let (decoded, _len): (AppId, usize) =
        decode_from_slice(&encoded, config::standard()).expect("bincode decode");

    assert_eq!(decoded, original);
    assert_eq!(decoded.as_bytes(), &bytes[..]);
}
```

Figure 1: Fuzzing harness added to `app_id.rs`

cargo-edit

`cargo-edit` identified one outdated dependency, `rand_core`, in the `threshold-signatures` codebase. Consider integrating Dependabot to automatically be aware of when dependencies have updates. After careful review, update the dependencies.

```
$ cargo upgrade --incompatible --dry-run
name            old req compatible latest  new req
====            ======= ========== ======  =======
ecdsa           0.16.8  0.16.9     0.16.9  0.16.9
elliptic-curve  0.13.5  0.13.8     0.13.8  0.13.8
frost-core      2.1.0   2.2.0      2.2.0   2.2.0
frost-ed25519   2.1.0   2.2.0      2.2.0   2.2.0
frost-secp256k1 2.1.0   2.2.0      2.2.0   2.2.0
k256            0.13.1  0.13.4     0.13.4  0.13.4
rand_core       0.6.4   0.6.4      0.9.3   0.9.3
rmp-serde       1.1.2   1.3.0      1.3.0   1.3.0
```

```
serde           1.0.175 1.0.226     1.0.226 1.0.226
serde_bytes     0.11.17 0.11.19     0.11.19 0.11.19
serde_json      1.0.143 1.0.145     1.0.145 1.0.145
subtle          2.5.0   2.6.1       2.6.1   2.6.1
```

Figure 2: This list shows dependencies and their versions for the `threshold-signatures` repository. The yellow highlighted versions indicate versions that are compatible with the most recent version, while the red highlighted crate indicates that the currently used version is incompatible with the most recent one.

```
cargo upgrade --incompatible --dry-run
name                     old req compatible latest  new req
====                     ======= ========== ======  =======
actix                    0.13.0  0.13.5     0.13.5  0.13.5
anyhow                   1.0.92  1.0.100    1.0.100 1.0.100
async-trait              0.1.83  0.1.89     0.1.89  0.1.89
axum                     0.7.9   0.7.9      0.8.4   0.8.4
backon                   1.5.1   1.5.2      1.5.2   1.5.2
borsh                    1.5.1   1.5.7      1.5.7   1.5.7
clap                     4.5.20  4.5.48     4.5.48  4.5.48
dstack-sdk-types         0.1.0   0.1.1      0.1.1   0.1.1
gcloud-sdk               0.26.2  0.26.4     0.28.2  0.28.2
getrandom                0.2.12  0.2.16     0.3.3   0.3.3
hex-literal              0.4.1   0.4.1      1.0.0   1.0.0
humantime                2.1.0   2.3.0      2.3.0   2.3.0
itertools                0.12.1  0.12.1     0.14.0  0.14.0
lru                      0.12.5  0.12.5     0.16.1  0.16.1
near-account-id          1.1.1   1.1.4      1.1.4   1.1.4
near-gas                 0.2.5   0.2.5      0.3.2   0.3.2
near-jsonrpc-client      0.15.1  0.15.1     0.18.0  0.18.0
near-jsonrpc-primitives  0.28.0  0.28.0     0.32.0  0.32.0
near-primitives          0.28.0  0.28.0     0.32.0  0.32.0
near-sdk                 5.15.1  5.17.2     5.17.2  5.17.2
prometheus               0.13.4  0.13.4     0.14.0  0.14.0
rand                     0.8.5   0.8.5      0.9.2   0.9.2
rand_xorshift            0.3     0.3.0      0.4.0   0.4
rcgen                    0.13.1  0.13.2     0.14.5  0.14.5
rocksdb                  0.21.0  0.21.0     0.24.0  0.24.0
rustls                   0.23.31 0.23.32    0.23.32 0.23.32
reqwest                  0.12.9  0.12.23    0.12.23 0.12.23
rstest                   0.25.0  0.25.0     0.26.1  0.26.1
schemars                 0.8.22  0.8.22     1.0.4   1.0.4
serde                    1.0.225 1.0.226    1.0.226 1.0.226
serde_json               1.0.132 1.0.145    1.0.145 1.0.145
serde_with               3.14.0  3.14.1     3.14.1  3.14.1
tempfile                 3.22.0  3.23.0     3.23.0  3.23.0
```

```
thiserror               2.0.12  2.0.16     2.0.16  2.0.16
time                    0.3.41  0.3.44     0.3.44  0.3.44
tokio                   1.41.0  1.47.1     1.47.1  1.47.1
tokio-util              0.7.12  0.7.16     0.7.16  0.7.16
tokio-rustls            0.26.1  0.26.3     0.26.3  0.26.3
tracing                 0.1.40  0.1.41     0.1.41  0.1.41
x509-parser             0.16.0  0.16.0     0.18.0  0.18.0

mpc-devnet
name                    old req compatible latest new req
====                    ======= ========== ====== =======
rand                    0.9.0   0.9.2      0.9.2  0.9.2
near-crypto             0.28.0  0.28.0     0.32.0 0.32.0
```

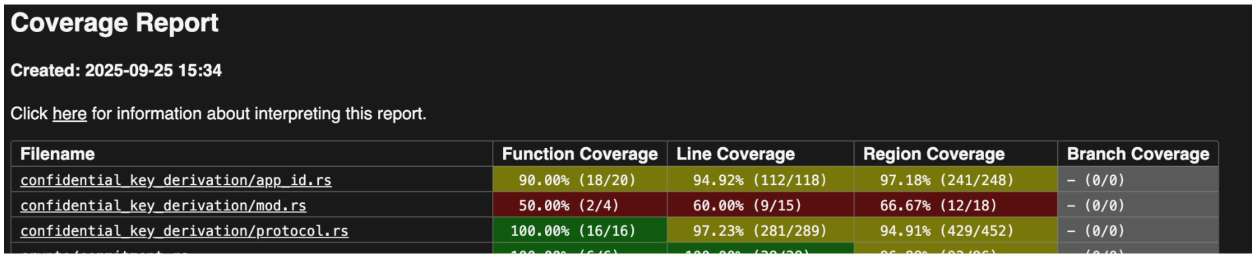
Figure 3: The dependency version comparison for the `mpc` *repository*

cargo-audit

`cargo-audit` identified one known vulnerability in the `wasmtime crate` (in the `mpc` repository). Additionally, it notes that there are six unmaintained crates in the dependencies: `wee_alloc`, `proc-macro-error`, `paste`, `instant`, `derivative`, and `atomic-polyfill`. Update the known vulnerability and run `cargo audit` on every commit as part of CI.

cargo-llvm-cov

We ran `cargo-llvm-cov` in the `threshold-signatures` repository and obtained a test code coverage report. Although there are a few untested functions in the `confidential-key-derivation` folder, these are only getters and from/into implementations.



Coverage Report

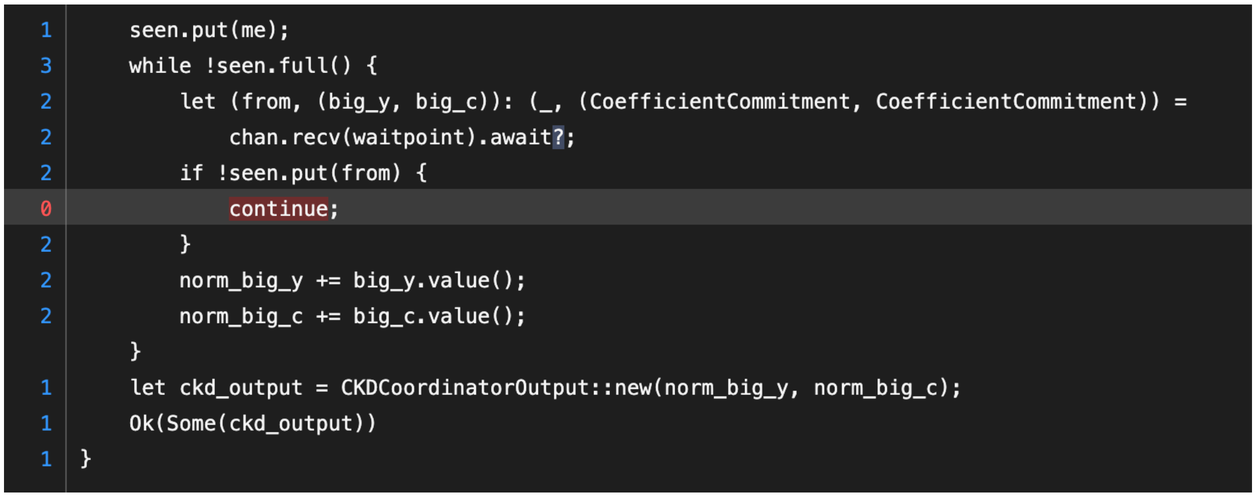
Created: 2025-09-25 15:34

Click here for information about interpreting this report.

Filename	Function Coverage	Line Coverage	Region Coverage	Branch Coverage
confidential_key_derivation/app_id.rs	90.00% (18/20)	94.92% (112/118)	97.18% (241/248)	- (0/0)
confidential_key_derivation/mod.rs	50.00% (2/4)	60.00% (9/15)	66.67% (12/18)	- (0/0)
confidential_key_derivation/protocol.rs	100.00% (16/16)	97.23% (281/289)	94.91% (429/452)	- (0/0)

Figure 4: Test code coverage report generated using `cargo llvm-cov --open`

The coordinator code has an untested branch when it sees a second message from the other participants in the channel. Since honest participants do not do this, an incorrect implementation of the protocol would be needed to test this branch.



```
1      seen.put(me);
3      while !seen.full() {
2          let (from, (big_y, big_c)): (_, (CoefficientCommitment, CoefficientCommitment)) =
2              chan.recv(waitpoint).await?;
2          if !seen.put(from) {
0              continue;
2          }
2          norm_big_y += big_y.value();
2          norm_big_c += big_c.value();
       }
1      let ckd_output = CKDCoordinatorOutput::new(norm_big_y, norm_big_c);
1      Ok(Some(ckd_output))
1  }
```

Figure 5: Test code coverage report highlighting a branch that is untested in `src/confidential_key_derivation/protocol.rs`

cargo-mutants

We ran `cargo-mutants` in the `confidential-key-derivation` folder. This tool will mutate the code and test whether the tests detect this change. The tool identified two timeouts in the tests:

- Altering the `do_ckd_participant` function to never write to the channel causes the coordinator to wait forever.
- Negating the `if` statement in the `run_ckd_protocol` function causes more coordinators than participants, and all will wait for participants that do not exist.

```
$ cargo mutants -f "src/confidential_key_derivation/*"
Found 64 mutants to test
ok       Unmutated baseline in 25.5s build + 24.3s test
 INFO Auto-set test timeout to 2m 2s
TIMEOUT  src/confidential_key_derivation/protocol.rs:42:5: replace
do_ckd_participant -> Result<CKDOutput, ProtocolError> with Ok(Default::default())
in 2.6s build + 122.0s test
TIMEOUT  src/confidential_key_derivation/protocol.rs:172:11: replace == with != in
run_ckd_protocol in 3.7s build + 122.0s test
    64 mutants tested in 15m 44s: 26 caught, 36 unviable, 2 timeouts
```

Figure 6: Results from running `cargo-mutants` *in the* `confidential_key_derivation` *folder*

Summary of Findings

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

ID	Title	Type	Severity
1	CKD protocol allows a malicious coordinator to control the derived key	Cryptography	**High**
2	CKD response handler lacks access controls	Access Controls	**High**
3	The index function panics when the participant is not in the list	Data Validation	**Medium**
4	AppId Borsh serialization function truncates the AppId bytes if the length is larger than u32::MAX	Data Validation	**Informational**
5	The CKD coordinator hangs forever waiting for all participants	Denial of Service	**Informational**
6	Polynomial generation function can panic or create a constant polynomial when called with a large degree	Data Validation	**Low**
7	Reliable broadcast indexes vectors to received indices	Data Validation	**Medium**
8	Resharing with a new participant and threshold equal to one will always fail	Data Validation	**Medium**
9	PolynomialCommitment deserialization function does not trim the coefficient list	Cryptography	**Informational**
10	CKDRequestStorage::get treats any broadcast receive error as fatal	Data Validation	**Informational**
11	Potential credential persistence in artifacts	Configuration	**Informational**
12	Unpinned external GitHub CI/CD action versions	Configuration	**Low**

13	Inconsistent handling of threshold in key refresh protocol	Cryptography	**Informational**
14	Unpinned versions and potential credential persistence in nearcore GitHub workflows	Configuration	**Low**
15	CKD protocol relies solely on MPC node secrets	Cryptography	**Informational**

Detailed Findings

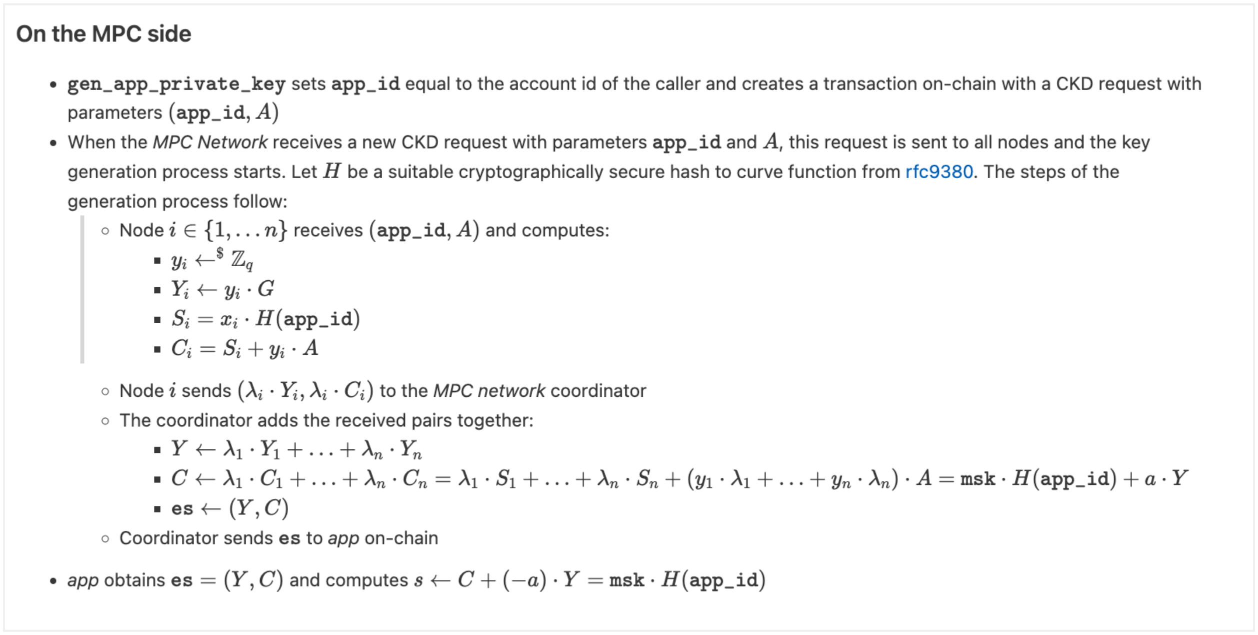
1. CKD protocol allows a malicious coordinator to control the derived key	
Severity: **High**	Difficulty: **Low**
Type: Cryptography	Finding ID: TOB-NEARCKD-1
Target: `threshold-signatures/docs/confidential_key_derivation.md`	

Description

A malicious coordinator can control the key derived in the CKD protocol, breaking the protocol's confidentiality.

The proposed CKD protocol enables an application running in a TEE to request the external derivation of a secret key using a set of NEAR contracts and a network of MPC nodes that run the CKD protocol. The goal is that this key is deterministic and known only to the application.

In the CKD protocol, a set of nodes interact to compute pairs of values, which are combined by the protocol coordinator, a randomly selected MPC node. However, since the protocol does not support the end receiver verifying the validity of the received values, the protocol coordinator can control the secret derived by the end application.



On the MPC side

- `gen_app_private_key` sets `app_id` equal to the account id of the caller and creates a transaction on-chain with a CKD request with parameters $(\mathtt{app\_id}, A)$
- When the *MPC Network* receives a new CKD request with parameters `app_id` and A, this request is sent to all nodes and the key generation process starts. Let H be a suitable cryptographically secure hash to curve function from rfc9380. The steps of the generation process follow:
  - Node $i \in \{1, \ldots n\}$ receives $(\mathtt{app\_id}, A)$ and computes:
    - $y_i \xleftarrow{\$} \mathbb{Z}_q$
    - $Y_i \leftarrow y_i \cdot G$
    - $S_i = x_i \cdot H(\mathtt{app\_id})$
    - $C_i = S_i + y_i \cdot A$
  - Node i sends $(\lambda_i \cdot Y_i, \lambda_i \cdot C_i)$ to the *MPC network* coordinator
  - The coordinator adds the received pairs together:
    - $Y \leftarrow \lambda_1 \cdot Y_1 + \ldots + \lambda_n \cdot Y_n$
    - $C \leftarrow \lambda_1 \cdot C_1 + \ldots + \lambda_n \cdot C_n = \lambda_1 \cdot S_1 + \ldots + \lambda_n \cdot S_n + (y_1 \cdot \lambda_1 + \ldots + y_n \cdot \lambda_n) \cdot A = \mathtt{msk} \cdot H(\mathtt{app\_id}) + a \cdot Y$
    - $\mathtt{es} \leftarrow (Y, C)$
  - Coordinator sends `es` to *app* on-chain
- *app* obtains $\mathtt{es} = (Y, C)$ and computes $s \leftarrow C + (-a) \cdot Y = \mathtt{msk} \cdot H(\mathtt{app\_id})$

Figure 1.1: CKD protocol description of the MPC nodes steps

The attack works by choosing `(Y, C) = (y·G, s + y·A)`, which causes the derived secret to be `s`, as `C – a·Y = s + y·A – a·Y = s + y·a·G – a·y·G = s`.

The lack of confidentiality of the end secret can also be determined by formally modeling the proposed protocol in ProVerif and querying for the secrecy of the derived key. In doing so, ProVerif immediately finds the attack described above. The ProVerif model and its output are included in appendix B.

In general, the proposed protocol does not guarantee its intended security goal of deriving a confidential and deterministic secret (observe how the coordinator can always choose a different secret for the same request).

Only when assuming fully honest participants, no man-in-the-middle attackers, and a vulnerability-free set of smart contracts that interact with the application and the set of MPC nodes could the protocol be considered adequate. However, as finding TOB-NEARCKD-2 shows, this assumption does not hold: due to missing access-control checks in the smart contract, any NEAR network account could execute the attack described here.

Exploit Scenario

An application wants to derive a key used to encrypt private data that is uploaded elsewhere using CKD. During the CKD protocol, the malicious coordinator ignores all MPC nodes' contributions and returns `(Y, C) = (y·G, s + y·A)`, causing the derived secret to be `s`, which is known to the coordinator. The coordinator uses this knowledge to decrypt the application's private data once it is uploaded.

Recommendations

Short term, consider adding a way to check for verifiability and a proof of correct computation to the CKD protocol using zero-knowledge proofs of discrete-log equality.

Long term, consider adapting well-studied and published protocols in the verifiable threshold OPRF literature, such as "Threshold PAKE with Security against Compromise of All Servers" and "Adaptively Secure Threshold Blind BLS Signatures and Threshold Oblivious PRF."

2. CKD response handler lacks access controls	
Severity: **High**	Difficulty: **Low**
Type: Access Controls	Finding ID: TOB-NEARCKD-2
Target: `mpc/crates/contract/src/lib.rs`	

Description

The `MpcContract`, which issues the CKD request to the MPC nodes and handles their response, lacks access controls when handling the response provided by the (supposedly) MPC nodes. This means that any user of the NEAR network could call the `respond_ckd` function and trigger the resolution of any CKD request with an arbitrary value.

The CKD workflow at the `MpcContract` works in steps:

1. Some external actor calls the `request_app_private_key` function; this triggers the MPC nodes to run the CKD protocol on the `app_id` corresponding to `env::predecessor_account_id()`, together with a public point and a `domainId`. This creates a `promise_yield_create` that states that once the promise is resumed with `promise_yield_resume`, it will call the `return_ck_and_clean_state_on_success` function.
2. Another public function in the contract, `respond_ckd`, receives the `CKDResponse` for an external party and issues the promise resolution with `promise_yield_resume`.
3. The `return_ck_and_clean_state_on_success` function is eventually executed, and the `CKDResponse` value is returned to the original caller.

The `respond_ckd` function lacks access controls, allowing any attacker to serve any value as the `CKDResponse`, regardless of its validity. Given that the proposed CKD protocol does not enforce any verifiability or robustness properties, it is impossible for the client application that is deployed inside the TEE, or the smart contract that receives the response, to determine whether the response is valid.

```
#[handle_result]
pub fn respond_ckd(&mut self, request: CKDRequest, response: CKDResponse) ->
Result<(), Error> {
    let signer = env::signer_account_id();
    log!("respond_ckd: signer={}, request={:?}", &signer, &request);

    if !self.protocol_state.is_running_or_resharing() {
        return Err(InvalidState::ProtocolStateNotRunning.into());
    }

    if !self.accept_requests {
        return Err(TeeError::TeeValidationFailed.into());
    }

    // First get the yield promise of the (potentially timed out) request.
    if let Some(YieldIndex { data_id }) = self.pending_ckd_requests.remove(&request)
{
        // Finally, resolve the promise. This will have no effect if the request
already timed.
        env::promise_yield_resume(&data_id,
&serde_json::to_vec(&response).unwrap());
        Ok(())
    } else {
        Err(InvalidParameters::RequestNotFound.into())
    }
}
```

Figure 2.1: `mpc/crates/contract/src/lib.rs#L487–L508`

Exploit Scenario

An attacker monitors the issuance of CKD requests and front runs the MPC nodes by immediately providing CKD responses. Using the attack described in TOB-NEARCKD-1, the attacker can act as the malicious coordinator and provide `(Y, C)` values for which they know the secret that is derived by the end application, breaking its confidentiality.

Recommendations

Short term, enforce access controls in the `CKD` response handler, enforcing that the MPC participant must be valid, attested, and active.

Long term, add verifiability properties to the CKD protocol, ensuring that the parties receiving the result from the protocol can verify the correctness of those values.

3. The index function panics when the participant is not in the list

Severity: **Medium**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-NEARCKD-3
Target: `threshold-signatures/src/participants.rs`, `threshold-signatures/src/protocol/echo_broadcast.rs`	

Description

The `ParticipantList::index` function directly accesses a `HashMap` using the bracket operator, causing the application to panic when queried with a participant that does not exist in the `HashMap`.

```
/// Return the index of a given participant.
///
/// Basically, the order they appear in a sorted list
pub fn index(&self, participant: &Participant) -> usize {
    self.indices[participant]
}
```

Figure 3.1: `threshold-signatures/src/participants.rs#L69–L74`

This vulnerable function is used in the `reliable_broadcast_receive_all` function, allowing a malicious participant to cause a runtime error in all other participants. Note how there is no validation of the `from` value before it is used in the `index` function:

```
// Am I handling a simulated vote sent by me to myself?
if !is_simulated_vote {
    // The recv should be failure-free
    // This translates to ignoring the received message when deemed wrong
    // types of the received answers are (Participant, (usize, MessageType))
    match chan.recv(wait).await {
        Ok(value) => (from, (sid, vote)) = value,
        _ => continue,
    };
};

is_simulated_vote = false;

match vote.clone() {
    // Receive send vote then echo to everybody
    MessageType::Send(data) => {
        // if the sender is not the one identified by the session id (sid)
        // or if the sender have already delivered a MessageType::Send message
        // then skip
        // the second condition prevents a malicious party starting the protocol
        // on behalf on somebody else
        if finish_send[sid] || sid != participants.index(&from) {
```

Figure 3.2: threshold-signatures/src/protocol/echo_broadcast.rs#L151–L172

Figure 3.3 shows a modification of the `do_broadcast_dishonest_consume_version_1` function that triggers the issue when running the `test_three_honest_one_dihonest` test.

```
pub async fn do_broadcast_dishonest_consume_version_1(
    mut chan: SharedChannel,
    participants: ParticipantList,
    me: Participant,
) -> Result<Vec<bool>, ProtocolError> {
    let wait_broadcast = chan.next_waitpoint();
    let sid = participants.index(&me);

    // malicious reliable broadcast send
    let vote_true = MessageType::Send(true);
    let vote_false = MessageType::Send(false);

    let other_participant = Participant::from(1337);
    chan.send_private(wait_broadcast, other_participant, &(&sid, &vote_true))?;

    for (cnt, p) in participants.others(me).enumerate() {
        if cnt >= participants.len() / 2 {
            chan.send_private(wait_broadcast, p, &(&sid, &vote_false))?;
        } else {
            chan.send_private(wait_broadcast, p, &(&sid, &vote_true))?;
        }
    }

    let vote_list =
        reliable_broadcast_receive_all(&chan, wait_broadcast, &participants, &me,
vote_false)
            .await?;
    let vote_list = vote_list.into_vec_or_none().unwrap();
    Ok(vote_list)
}

// thread 'protocol::echo_broadcast::test::test_three_honest_one_dihonest' panicked at
src/protocol/mod.rs:158:35:
// no entry found for key
// stack backtrace:
//    0: __rustc::rust_begin_unwind
//              at /rustc/library/std/src/panicking.rs:697:5
//    1: core::panicking::panic_fmt
//              at /rustc/library/core/src/panicking.rs:75:14
//    2: core::panicking::panic_display
//              at /rustc/library/core/src/panicking.rs:269:5
//    3: core::option::expect_failed
//              at /rustc/library/core/src/option.rs:2049:5
//    4: core::option::Option<T>::expect
//              at /lib/rustlib/src/rust/library/core/src/option.rs:958:21
//    5: <std::collections::hash::map::HashMap<K,V,S> as core::ops::index::Index<&Q>>::index
```

Figure 3.3: Modification of the test helper that will trigger the runtime panic in the broadcast handling, and error trace from running the `test_three_honest_one_dihonest` test (`threshold-signatures/src/protocol/echo_broadcast.rs`)

Exploit Scenario

A malicious protocol participant causes the other participants to crash by sending an invalid `Participant` value that is received by others during the `reliable_broadcast_receive_all` function.

Recommendations

Short term, use the `get` function to access the `HashMap` instead. This will prevent the application from crashing and allow proper error handling when invalid participants are encountered.

Long term, add unit tests to the `ParticipantList` functions.

4. AppId Borsh serialization function truncates the AppId bytes if the length is larger than u32::MAX

Severity: **Informational**	Difficulty: **N/A**
Type: Data Validation	Finding ID: TOB-NEARCKD-4
Target: `threshold-signatures/src/confidential_key_derivation/app_id.rs`	

Description

The `Borsh` serialization function casts the length of the `AppId` bytes to u32, causing data truncation at deserialization time if more than `u32::MAX` bytes are provided.

```
impl BorshSerialize for AppId {
    fn serialize<W: std::io::Write>(&self, writer: &mut W) -> std::io::Result<()> {
        // serialize as Vec<u8>
        let bytes: &[u8] = &self.0;
        borsh::BorshSerialize::serialize(&(bytes.len() as u32), writer)?;
        writer.write_all(bytes)
    }
}

impl BorshDeserialize for AppId {
    fn deserialize_reader<R: std::io::Read>(reader: &mut R) -> std::io::Result<Self>
{
        let len = u32::deserialize_reader(reader)? as usize;
        let mut buf = vec![0u8; len];
        reader.read_exact(&mut buf)?;
        Ok(AppId::from(buf))
    }
}
```

Figure 4.1: *`threshold-signatures/src/confidential_key_derivation/app_id.rs#L86–L102`*

The serialization format encodes the length of the data, followed by the data: `bytes.len()` as u32`|bytes`. This causes an `AppId` of size `u32::MAX` + 1 to be serialized as `0|bytes`. At deserialization time, the result will be the empty vector, as the serialized length is zero.

The following test exercises this edge case:

```
    #[test]
    fn borch_larger_than_u32_max() {
        let large_size = (u32::MAX as usize) + 1;
        let large_bytes = vec![0u8; large_size];
        let app_id = AppId::new(large_bytes);

        let mut buf = vec![];
        let result = borsh::BorshSerialize::serialize(&app_id, &mut buf);

        assert!(result.is_ok(), "Serialization works without errors");

        let decoded = AppId::deserialize_reader(&mut buf.as_slice()).unwrap();

        assert!(decoded.as_bytes().len() == 0, "Length equals zero due to
truncation");

        if decoded != app_id {
            panic!("Decoded does not match original");
        }
    }
```

Figure 4.2: Test exercising the truncation issue in `Borsh` serialization and deserialization

This issue has an informational severity rating because, under normal circumstances, the `AppId` viewed by the CKD protocol is computed from the `receipt.predecessor_id`, which should be a validated and length-checked `AccountId`.

Recommendations

Short term, implement a checked conversion to `u32` with `u32::try_from`, returning an error if the conversion fails. Fix all Clippy's potential truncation warnings identified using the `cast_possible_truncation` rule.

Long term, add tests for integer type edge cases, and unify the `AppId` type with the type used in the `mpc` library, `AccountId`, which has in-built validation.

5. The CKD coordinator hangs forever waiting for all participants	
Severity: **Informational**	Difficulty: **N/A**
Type: Denial of Service	Finding ID: TOB-NEARCKD-5
Target: `threshold-signatures/src/confidential_key_derivation/protocol.rs`	

Description

The implementation of the CKD coordinator loops forever, waiting for the contribution of all other participants. This could allow the TEE host of an MPC participant to indefinitely halt the execution of the CKD protocol and the coordinator machine.

The NEAR One team has stated that such timeout handling is part of the libraries that call into the CKD codebase.

```
seen.put(me);
while !seen.full() {
    let (from, (big_y, big_c)): (_, (CoefficientCommitment, CoefficientCommitment))
=
        chan.recv(waitpoint).await?;
    if !seen.put(from) {
        continue;
    }
    norm_big_y += big_y.value();
    norm_big_c += big_c.value();
}
let ckd_output = CKDCoordinatorOutput::new(norm_big_y, norm_big_c);
Ok(Some(ckd_output))
```

Figure 5.1: `threshold-signatures/src/confidential_key_derivation/protocol.rs#L88–L99`

Recommendations

Short term, add documentation to the CKD codebase informing library users that its callers will need to ensure correct timeout setup.

Long term, add tests to the library that calls into CKD, exercising correct handling of the timeout behavior.

6. Polynomial generation function can panic or create a constant polynomial when called with a large degree

Severity: **Low**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-NEARCKD-6

Target:
- `threshold-signatures/src/crypto/polynomials.rs`
- `threshold-signatures/src/ecdsa/robust_ecdsa/presign.rs`
- `threshold-signatures/src/ecdsa/ot_based_ecdsa/triples/generation.rs`
- `threshold-signatures/src/generic_dkg.rs`

Description

The `generate_polynomial` function returns a constant polynomial when the input degree is `usize::MAX` due to an integer overflow, and will panic due to exceedingly large capacity in the coefficient vector allocation for degrees between `isize::MAX` and `usize::MAX-1`.

The codebase allows paths that call the `generate_polynomial` function with such large degrees due to missing validations on the threshold value and other arithmetic underflows (e.g., via the `generate_triple_many` function).

```
/// Creates a random polynomial p of the given degree
/// and sets p(0) = secret
/// if the secret is not given then it is picked at random
pub fn generate_polynomial(
    secret: Option<Scalar<C>>,
    degree: usize,
    rng: &mut impl CryptoRngCore,
) -> Result<Self, ProtocolError> {
    let poly_size = degree + 1;
    let mut coefficients = Vec::with_capacity(poly_size);
    // insert the secret share if exists
    let secret = secret.unwrap_or_else(|| <C::Group as Group>::Field::random(rng));

    coefficients.push(secret);
    for _ in 1..poly_size {
        coefficients.push(<C::Group as Group>::Field::random(rng));
    }
    // fails only if:
    // * polynomial is of degree 0 and the constant term is 0
    // * polynomial degree is the max of usize, and so degree + 1 is 0
    // such cases never happen in a classic (non-malicious) implementations
    Self::new(coefficients)
}
```

Figure 6.1: `threshold-signatures/src/crypto/polynomials.rs#L50-L72`

```
if args.threshold > participants.len() {
    return Err(InitializationError::BadParameters(
        "threshold must be less than or equals to participant count".to_string(),
    ));
}

// if 2 * args.threshold + 1 > participants.len()
// this complex way prevents overflowing
if args
    .threshold
    .saturating_mul(2)
    .checked_add(1)
    .ok_or_else(|| {
        InitializationError::BadParameters(
            "2*threshold+1 must be less than usize::MAX".to_string(),
        )
    })?
    > participants.len()
{
    return Err(InitializationError::BadParameters(
        "2*threshold+1 must be less than or equals to participant
count".to_string(),
    ));
}

let participants =

ParticipantList::new(participants).ok_or(InitializationError::DuplicateParticipants)
?;

if !participants.contains(me) {
    return Err(InitializationError::MissingParticipant {
        role: "self",
        participant: me,
    });
};

let ctx = Comms::new();
let fut = do_presign(ctx.shared_channel(), participants, me, args, rng);
```

Figure 6.2: Missing lower bounds when performing validation (`threshold-signatures/src/ecdsa/robust_ecdsa/presign.rs#L42-L77`*)*

```
// Spec 1.1
if threshold > participants.len() {
    return Err(InitializationError::ThresholdTooLarge {
        threshold,
        max: participants.len(),
    });
}

let participants =

ParticipantList::new(participants).ok_or(InitializationError::DuplicateParticipants)
?;

let ctx = Comms::new();
let fut = do_generation(ctx.clone(), participants, me, threshold, rng);
Ok(make_protocol(ctx, fut))
```

Figure 6.3: `threshold-signatures/src/ecdsa/ot_based_ecdsa/triples/generation.rs#L1132–L1145`

```
pub(crate) fn assert_keygen_invariants(
    participants: &[Participant],
    me: Participant,
    threshold: usize,
) -> Result<ParticipantList, InitializationError> {
    // need enough participants
    if participants.len() < 2 {
        return Err(InitializationError::NotEnoughParticipants {
            participants: participants.len(),
        });
    };

    // validate threshold
    if threshold > participants.len() {
        return Err(InitializationError::ThresholdTooLarge {
            threshold,
            max: participants.len(),
        });
    }
```

Figure 6.4: `threshold-signatures/src/generic_dkg.rs#L528–L546`

```
async fn do_generation(
    comms: Comms,
    participants: ParticipantList,
    me: Participant,
    threshold: usize,
    mut rng: impl CryptoRngCore + Send + 'static,
) -> Result<TripleGenerationOutput, ProtocolError> {
    let mut chan = comms.shared_channel();
    let mut transcript = create_transcript(&participants, threshold)?;

    // Spec 1.2
    let e = Polynomial::generate_polynomial(None, threshold - 1, &mut rng)?;
    let f = Polynomial::generate_polynomial(None, threshold - 1, &mut rng)?;
    // Spec 1.3
    // We will generate a poly of degree threshold - 2 then later extend it with
identity.
    // This is to prevent serialization from failing
    let mut l = Polynomial::generate_polynomial(None, threshold - 2, &mut rng)?;
```

Figure 6.5:
`threshold-signatures/src/ecdsa/ot_based_ecdsa/triples/generation.rs`

Exploit Scenario
An attacker participating in a protocol convinces other participants to use a threshold of 0 in the triple generation protocol, leading all participants to crash due to a capacity overflow when allocating the polynomial coefficients vector.

Recommendations
Short term, add checks to ensure that the `generate_polynomial` function returns an error if the degree is larger than `isize::MAX`. Ensure that checked arithmetic is used in the function and in threshold adjustments (e.g., in the `generate_triple_many` function). Add lower bounds checks to the threshold value in the validation functions.

Long term, add fuzz testing and `kani` proofs to simple functions with primitive types; review all uses of unchecked arithmetic in the codebase.

7. Reliable broadcast indexes vectors to received indices

Severity: **Medium**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-NEARCKD-7
Target: `threshold-signatures/src/protocol/echo_broadcast.rs`	

Description

The `reliable_broadcast_receive_all` function accesses vectors at arbitrary indices, allowing a malicious protocol participant to induce a panic in any protocol participant.

Once received, the `sid` value is not checked to be within bounds before it is used to access the vectors.

```
    match chan.recv(wait).await {
          Ok(value) => (from, (sid, vote)) = value,
          _ => continue,
     };
};

is_simulated_vote = false;

match vote.clone() {
     // Receive send vote then echo to everybody
     MessageType::Send(data) => {
         // if the sender is not the one identified by the session id (sid)
         // or if the sender have already delivered a MessageType::Send message
         // then skip
         // the second condition prevents a malicious party starting the protocol
         // on behalf on somebody else
         if finish_send[sid] || sid != participants.index(&from) {
             continue;
```

Figure 7.1: `threshold-signatures/src/protocol/echo_broadcast.rs#L156–L173`

Exploit Scenario

A malicious protocol participant sends a large `sid` value that is not validated in the `reliable_broadcast_receive_all` function, causing all other participants to panic.

Recommendations

Short term, add validation of all other participants' contributions to the protocol, including the `sid` and `from` values, and ensure the code includes proper error resolution.

Long term, add tests exercising the robust handling of malformed or malicious messages in the echo broadcast protocol.

8. Resharing with a new participant and threshold equal to one will always fail

Severity: **Medium**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-NEARCKD-8
Target: `threshold-signatures/src/generic_dkg.rs`	

Description

The reshare protocol will always fail when the threshold is one and a new participant joins the protocol.

When a new participant is participating in the reshare protocol, it sets its secret to zero and creates a polynomial of degree `threshold - 1`. However, since the `Polynomial::new` function returns an error for the constant zero polynomial, this resharing protocol will always fail.

```
/// Performs the heart of DKG, Reshare and Refresh protocols
async fn do_keyshare<C: Ciphersuite>(
    mut chan: SharedChannel,
    participants: ParticipantList,
    me: Participant,
    threshold: usize,
    secret: Scalar<C>,
    old_reshare_package: Option<(VerifyingKey<C>, ParticipantList)>,
    rng: &mut impl CryptoRngCore,
) -> Result<KeygenOutput<C>, ProtocolError> {
    let mut all_full_commitments = ParticipantMap::new(&participants);
    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
    rng.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;
    // the degree of the polynomial is threshold - 1
    let secret_coefficients =
        Polynomial::<C>::generate_polynomial(Some(secret), threshold - 1, rng)?;
```

Figure 8.1: `threshold-signatures/src/generic_dkg.rs#L328–L358`

```
/// reshares the keyshares between the parties and allows changing the threshold
#[allow(clippy::too_many_arguments)]
pub(crate) async fn do_reshare<C: Ciphersuite>(
    chan: SharedChannel,
    participants: ParticipantList,
    me: Participant,
    old_threshold: usize,
    old_signing_key: Option<SigningShare<C>>,
    old_public_key: VerifyingKey<C>,
    old_participants: ParticipantList,
    mut rng: impl CryptoRngCore,
) -> Result<KeygenOutput<C>, ProtocolError> {
    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
                .lagrange::<C>(me)
                .map(|lambda| lambda * x_i.to_scalar())
        })
        .transpose()?
```

Figure 8.2: `threshold-signatures/src/generic_dkg.rs#L562–L582`

Exploit Scenario

A new participant joins the MPC nodes, where the threshold is set to one. However, the new participant can never generate the secret polynomial, and the protocol will always fail.

Recommendations

Short term, add validation to the threshold value and add checked arithmetic to operations on it. Add edge case tests exercising cases where the operations should fail and where they should still succeed.

Long term, formally specify the resharing and refresh protocols, ensuring that all necessary checks are described. Then, annotate the implementation with the specification steps, ensuring that no step is missing from the implementation.

9. PolynomialCommitment deserialization function does not trim the coefficient list

Severity: **Informational**	Difficulty: **N/A**
Type: Cryptography	Finding ID: TOB-NEARCKD-9

Target:
- `threshold-signatures/src/crypto/polynomials.rs`
- `threshold-signatures/docs/ecdsa/triples.md`
- `threshold-signatures/src/ecdsa/ot_based_ecdsa/triples/generation.rs`

Description

The `PolynomialCommitment::deserialize` function allows trailing identity elements in its representation, causing functions that rely on a trimmed commitment list, such as the `degree` function, to return incorrect results. This could be problematic in the Cait-Sith implementation if the serialized data could contain the identity point, but this is not the case.

```
// Deserialization enforcing non-empty vecs and non all-identity
PolynomialCommitments
impl<'de, C: Ciphersuite> Deserialize<'de> for PolynomialCommitment<C> {
    fn deserialize<D>(deserializer: D) -> Result<Self, D::Error>
    where
        D: Deserializer<'de>,
    {
        let coefficients =
Vec::<CoefficientCommitment<C>>::deserialize(deserializer)?;
        if coefficients.is_empty() {
            Err(serde::de::Error::custom("Polynomial must not be empty"))
        } else {
            // counts the number of successive identity elements on the highest
            // degree coefficients and aborts if the committed polynomial is the
identity
            let is_identity = coefficients
                .iter()
                .rev()
                .all(|x| x.value() == C::Group::identity());
            if is_identity {
                return Err(serde::de::Error::custom(
                    "Polynomial must not be the identity",
                ));
            }
            Ok(PolynomialCommitment { coefficients })
        }
    }
}
```

Figure 9.1: `threshold-signatures/src/crypto/polynomials.rs#L352-L376`

Recommendations

Short term, use the `PolynomialCommitment::new` function in the deserialization routine, ensuring that all created elements of that type enforce the same invariants. Optionally, given that the identity point cannot be serialized, simplify the logic so that the deserialized data is used immediately and is validated only to ensure that it is non-empty.

Long term, add tests that exercise the polynomial commitment type created from a serialized/deserialized round-trip.

10. CKDRequestStorage::get treats any broadcast receive error as fatal

Severity: **Informational**	Difficulty: **N/A**
Type: Data Validation	Finding ID: TOB-NEARCKD-10
Target: `mpc/crates/node/src/storage.rs`	

Description

Tokio's broadcast could return `RecvError::Lagged`; the current implementation would discard such errors instead of fetching the earliest received value still in the channel.

```
/// Blocks until a ckd request with given id is present, then returns it.
/// This behavior is necessary because a peer might initiate computation for a ckd
/// request before our indexer has caught up to the request. We need proof of the
request
/// from on-chain in order to participate in the computation.
pub async fn get(&self, id: CKDId) -> Result<CKDRequest, anyhow::Error> {
    let key = borsh::to_vec(&id)?;
    let mut rx = self.add_sender.subscribe();
    if let Some(request_ser) = self.db.get(DBCol::CKDRequest, &key)? {
        return Ok(serde_json::from_slice(&request_ser)?);
    }
    loop {
        let added_id = match rx.recv().await {
            Ok(added_id) => added_id,
            Err(e) => {
                metrics::CKD_REQUEST_CHANNEL_FAILED.inc();
                return Err(anyhow::anyhow!("Error in ckd_request channel recv, {}",
e));
            }
```

Figure 10.1: `mpc/crates/node/src/storage.rs#L114–L117`

Recommendations

Short term, either implement explicit handling of lagged cases or make it explicit that this case is being ignored in a comment.

Long term, add tests that ensure the correct handling of a lagged receive.

11. Potential credential persistence in artifacts	
Severity: **Informational**	Difficulty: **N/A**
Type: Configuration	Finding ID: TOB-NEARCKD-11
Target: `mpc` and `threshold-signatures` repositories' workflows	

Description

The default behavior of the `actions/checkout` GitHub action is to persist credentials, which means that the GitHub token is written to the local repository directory. This allows the action to execute Git commands that require authentication. The credentials are stored in `.git/config` files in the local repository directory, where they could be inadvertently included in workflow artifacts or be accessed by subsequent workflow steps.

```
steps:
  - name: Checkout repository
    uses: actions/checkout@v4
    with:
      ref: ${{ github.event.inputs.build-ref }}

  - name: Login to Docker Hub
    uses: docker/login-action@v3
    with:
      username: ${{ secrets.DOCKERHUB_USERNAME }}
      password: ${{ secrets.DOCKERHUB_TOKEN }}

  - name: Build and publish images
    shell: bash
    run: |
      ./deployment/build-images.sh --launcher --push
```

Figure 11.1: *`.github/workflows/docker_launcher_release.yml#L23–L38`*

Recommendations

Short term, add `persist-credentials: false` to checkout actions that do not require privileged Git operations.

Long term, add `zizmor` to the CI/CD pipeline and fix all of its reported findings.

References

- ArtiPACKED: Hacking Giants through a Race Condition in GitHub Actions Artifacts
- `zizmor` ArtiPACKED documentation

12. Unpinned external GitHub CI/CD action versions	
Severity: **Low**	Difficulty: **High**
Type: Configuration	Finding ID: TOB-NEARCKD-12
Target: `mpc` and `threshold-signatures` repositories' workflows	

Description

Several GitHub Actions workflows in the `mpc` and `threshold-signatures` repositories use third-party actions pinned to a tag or branch name instead of a full commit SHA, as recommended by GitHub. This configuration enables repository owners to silently modify the actions. A malicious actor could use this ability to tamper with an application release or leak secrets.

The following actions are owned by GitHub organizations or individuals that are not affiliated directly with NEAR One:

- `Warpbuilds/build-push-action@v6`
- `WarpBuilds/rust-cache@v2`
- `docker/login-action@v3`
- `amannn/action-semantic-pull-request@v5`
- `astral-sh/ruff-action@v3`
- `WarpBuilds/setup-python@v5`
- `streetsidesoftware/cspell-action@v6`
- `peter-evans/create-issue-from-file@v4`
- `aws-actions/configure-aws-credentials@v4`
- `google-github-actions/auth@v2`
- `fregante/setup-git-user@v2`
- `taiki-e/install-action@cargo-llvm-cov`
- `codecov/codecov-action@v5`

```
- name: Build MPC Docker image
  uses: Warpbuilds/build-push-action@v6
  with:
    context: .
    profile-name: "mpc-image-builder"
    file: deployment/Dockerfile-node
    tags: test_image_tag_ci
    load: true
```

Figure 12.1: `.github/workflows/ci.yml#L27–L34`

Exploit Scenario
An attacker gains unauthorized access to the account of a GitHub action owner. The attacker manipulates the action's code to steal GitHub credentials and compromise the repository.

Recommendations
Short term, pin each third-party action to a specific full-length commit SHA, as recommended by GitHub. Additionally, configure Dependabot to update the actions' commit SHAs after reviewing their available updates.

Long term, add `zizmor` to the CI/CD pipeline.

13. Inconsistent handling of threshold in key refresh protocol

Severity: **Informational**	Difficulty: **N/A**
Type: Cryptography	Finding ID: TOB-NEARCKD-13
Target: `threshold-signatures/src/generic_dkg.rs`, `threshold-signatures/src/lib.rs`	

Description

The key refresh protocol ensures that the old and new thresholds are equal, but the functions used by the protocol do not validate this assumption, and there is a risk that changing the behavior of the top-level protocol could introduce subtle security issues.

The refresh protocol looks like this:

```
{
    if old_signing_key.is_none() {
        return Err(InitializationError::BadParameters(format!(
            "The participant {me:?} is running refresh without an old share",
        )));
    }
    let comms = Comms::new();
    let threshold = old_threshold;
    let (participants, old_participants) = reshare_assertions::<C>(
        old_participants,
        me,
        threshold,
        old_signing_key,
        threshold,
        old_participants,
    )?;
    let fut = do_reshare(
        comms.shared_channel(),
        participants,
        me,
        threshold,
        old_signing_key,
        old_public_key,
        old_participants,
        rng,
    );
    Ok(make_protocol(comms, fut))
}
```

Figure 13.1: `threshold-signatures/src/lib.rs#L97–L124`

The equality of `threshold` and `old_threshold` obscures an inconsistency in the lower layers of the protocol as currently written.

The validation steps performed in `reshare_assertions` include the following:

- There must be at least two participants in the new participant list.
- The new threshold cannot be larger than the new participant list.
- The participant list must contain the local participant ("me").
- The old participant list cannot be smaller than the old threshold, in order to recover the key to reshare it.
- The old participant list must contain the local participant.

However, there is no logic comparing the new threshold with the old threshold in `reshare_assertions` nor in `do_reshare`. This consistency is enforced only by the top-level `refresh` function.

A previous version of the `frost-core` library was impacted by a missing constraint across the old and new thresholds, which could render new signing operations inaccessible or reduce the security of participants' shares.

Although there is presently no immediate security risk with the current implementation, if the top-level refresh function is ever updated to support a different threshold between the old and new participant lists, this inconsistent handling of thresholds in the lower-level functions could introduce availability or confidentiality issues, as seen in `frost-core`.

Recommendations

Short term, document that threshold congruence is deliberate and that removing this property could have security consequences, so that it is not accidentally changed without the developer being aware.

Long term, update the APIs to not tolerate different threshold values during key refresh if they are not intended to be changed.

14. Unpinned versions and potential credential persistence in nearcore GitHub workflows	
Severity: **Low**	Difficulty: **High**
Type: Configuration	Finding ID: TOB-NEARCKD-14
Target: `mpc/libs/nearcore/.github/workflows/`	

Description

The GitHub workflows from the `nearcore` repository also suffer from the persistent credential issue and unpinned versions reported in TOB-NEARCKD-11 and TOB-NEARCKD-12.

Recommendations

Short term, add `zizmor` to all of the NEAR projects, starting with those that are public and the workflows that run on pull requests.

Long term, ensure that the “Approval for running fork pull request workflows from contributors” configuration is set to “Require approval for all external contributors” in all projects’ settings.

15. CKD protocol relies solely on MPC node secrets

Severity: **Informational**	Difficulty: **N/A**
Type: Cryptography	Finding ID: TOB-NEARCKD-15
Target: `threshold-signatures/docs/confidential_key_derivation.md`	

Description

The CKD protocol derives application keys using only private data from MPC nodes, without incorporating any secret material from the requesting application. The derived key is equivalent to `s = msk · H(app_id)`, where `msk` is the master secret key held by the MPC network and `app_id` is the public authenticated identifier provided by the application's smart contract.

Since the derived key depends exclusively on the MPC nodes' secrets and the public `app_id`, it is not necessary to attack the application to compute the key; attacks that compromise the MPC infrastructure are sufficient. Such attacks could be possible via smart-contract vulnerabilities (such as TOB-NEARCKD-2) or TEE attacks. Using a threshold OPRF scheme would force the attacker to also attack the application environment to learn its secret key.

Recommendations

Short term, document that application keys depend solely on MPC infrastructure security in the threat model section of the CKD specification. Perform a comparative analysis of the security guarantees between CKD and the considered alternatives.

Long term, consider deriving an application-specific key using Dstack's KMS in KMS mode. Then, if necessary, use one of the published OPRF schemes to derive a joint key using the MPC nodes' master secret key and the application's KMS-derived key. This would ensure that the derived key depends on both the MPC infrastructure and the application's security, and that no single party (the application or an attacker controlling a majority of the MPC nodes) could derive the derived key.

References

- Threshold PAKE with Security against Compromise of All Servers
- A Fully-Adaptive Threshold Partially-Oblivious PRF

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. ProVerif Model of the CKD Protocol

ProVerif is an automatic cryptographic protocol verifier in the symbolic (Dolev-Yao) model.

Figure B.1 shows a ProVerif model of the application side of the CKD protocol, where the application publishes `A = a G`, receives `(Y, C)`, and computes the derived secret `s = C - a Y`. This model is sufficient to identify the lack of confidentiality of the derived secret reported in TOB-NEARCKD-1.

```
type sk.
type pt.

(* Public constructors *)
fun multG(sk): pt.
fun add(pt, pt): pt.
fun scalarmult(sk, pt): pt.
fun sub(pt, pt): pt.

(* Algebraic properties *)
equation forall s, t: sk; scalarmult(s, multG(t)) = scalarmult(t, multG(s)).
equation forall x,y: pt; sub(add(x, y), y) = x.

(* Public channel (attacker controls it) *)
free c: channel.

(* The App: publish A = a·G, receive (Y, C), compute s = C - a·Y *)
let App =
  new a: sk;
  let A = multG(a) in
  out(c, A);                (* app's public key A = a·G *)
  in(c, (Y: pt, C: pt));    (* coordinator (or attacker) supplies (Y, C) *)
  let s = sub(C, scalarmult(a, Y)) in
  0.

(* secrecy of s *)
query secret s.

process
  (* Only the App; the Dolev-Yao attacker can use c *)
  App

(* Output of running `proverif ckd.pv`:

The attack is as follows:
    new a: sk creating a_1 at {1}

    out(c, ~M) with ~M = multG(a_1) at {3}

    in(c, (multG(a_2),add(a_3,scalarmult(a_2,~M)))) with add(a_3,scalarmult(a_2,~M))
= add(a_3,scalarmult(a_1,multG(a_2))) at {4}
```

```
    event
s_contains(sub(add(a_3,scalarmult(a_1,multG(a_2))),scalarmult(a_1,multG(a_2)))) at
{6} (goal)

    The event s_contains(a_3) is executed at {6}.
    The attacker has the message a_3.
    --------------------------------------------------------------
    Verification summary:
    Query secret s is false.
    --------------------------------------------------------------
*)
```

Figure B.1: ProVerif model of the CKD protocol, and its output demonstrating the confidentiality attack

C. Code Quality Issues

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

- **Use of `unwrap_or` with function calls.** Arguments passed to `unwrap_or` are eagerly evaluated and may lead to unnecessary computation. Since, in this case, a function is evaluated, we recommend using `unwrap_or_else` instead, as it will evaluate this function only if needed, preventing unnecessary computation. The codebase contains one instance of this pattern in the `threshold-signatures` repository and 14 in the `mpc` repository. Use Clippy's `or_fun_call` rule to find and fix all instances of this pattern.

```
let secret = old_signing_key
    .map(|x_i| {
        intersection
            .lagrange::<C>(me)
            .map(|lambda| lambda * x_i.to_scalar())
    })
    .transpose()?
    .unwrap_or(<C::Group as Group>::Field::zero());
```

Figure C.1: Use of `unwrap_or` with a function call
(`threshold-signatures/src/generic_dkg.rs#L576–L583`)

- **CKD implementation duplicates code.** Both the participants and the coordinator perform the same initial steps, but the code is duplicated in both functions. Refactoring the code into a separate function will ensure that the code is indeed the same and that if changes are made to one implementation, all implementations will see those changes.

```
#[allow(clippy::too_many_arguments)]
async fn do_ckd_participant(
    mut chan: SharedChannel,
    participants: ParticipantList,
    coordinator: Participant,
    me: Participant,
    private_share: SigningShare,
    app_id: &AppId,
    app_pk: VerifyingKey,
    rng: &mut impl CryptoRngCore,
) -> Result<CKDOutput, ProtocolError> {
    // y <- ZZq* , Y <- y * G
    let (y, big_y) = Secp256K1Sha256::generate_nonce(rng);
    // H(app_id) when H is a random oracle
    let hash_point = hash2curve(app_id)?;
    // S <- x . H(app_id)
    let big_s = hash_point * private_share.to_scalar();
    // C <- S + y . A
    let big_c = big_s + app_pk.to_element() * y;
    // Compute  λi := λi(0)
    let lambda_i = participants.lagrange::<Secp256K1Sha256>(me)?;
    // Normalize Y and C into  (λi . Y , λi . C)
    let norm_big_y = CoefficientCommitment::new(big_y * lambda_i);
    let norm_big_c = CoefficientCommitment::new(big_c * lambda_i);

    let waitpoint = chan.next_waitpoint();
```

Figure C.2: Excerpt of `protocol.rs` (`threshold-signatures/src/confidential_key_derivation/protocol.rs#L30–L55`)

```
async fn do_ckd_coordinator(
    mut chan: SharedChannel,
    participants: ParticipantList,
    me: Participant,
    private_share: SigningShare,
    app_id: &AppId,
    app_pk: VerifyingKey,
    rng: &mut impl CryptoRngCore,
) -> Result<CKDOutput, ProtocolError> {
    // y <- ZZq* , Y <- y * G
    let (y, big_y) = Secp256K1Sha256::generate_nonce(rng);
    // H(app_id) when H is a random oracle
    let hash_point = hash2curve(app_id)?;
    // S <- x . H(app_id)
    let big_s = hash_point * private_share.to_scalar();
    // C <- S + y . A
    let big_c = big_s + app_pk.to_element() * y;
    // Compute  λi := λi(0)
    let lambda_i = participants.lagrange::<Secp256K1Sha256>(me)?;
    // Normalize Y and C into  (λi . Y , λi . C)
    let mut norm_big_y = big_y * lambda_i;
    let mut norm_big_c = big_c * lambda_i;

    // Receive everyone's inputs and add them together
    let mut seen = ParticipantCounter::new(&participants);
    let waitpoint = chan.next_waitpoint();
```

Figure C.3: Excerpt of `protocol.rs` (`threshold-signatures/src/confidential_key_derivation/protocol.r#L61–L86`)

- **Use `participants.len()` instead of `participants.participants.len()`.** The `ParticipantList` type offers the `len` method, which should be used here.

```
pub fn new(participants: &'a ParticipantList) -> Self {
    // We could also require a T: Clone bound instead of doing this
```

```
initialization manually.
    let size = participants.participants.len();
```

Figure C.4: threshold-signatures/src/participants.rs#L138–L140

- **Missing use of fill or vec! initialization macro.** Replace the manual vector initialization with a call to the fill function, or create the vector with the vec! macro.

```
pub fn new(participants: &'a ParticipantList) -> Self {
    // We could also require a T: Clone bound instead of doing this initialization
manually.
    let size = participants.participants.len();
    let mut data = Vec::with_capacity(size);
    for _ in 0..size {
        data.push(None);
    }
```

Figure C.5: threshold-signatures/src/participants.r#L138–L144

- **Use of the handle_result attribute in methods that do not return a Result type.** The sign and request_app_private_key methods do not return a Result type but have the handle_result attribute.

```
#[handle_result]
#[payable]
pub fn sign(&mut self, request: SignRequestArgs) {
```

Figure C.6: mpc/crates/contract/src/lib.rs#L133–L135

```
#[handle_result]
#[payable]
pub fn request_app_private_key(&mut self, request: CKDRequestArgs) {
```

Figure C.7: mpc/crates/contract/src/lib.rs#L299–L303

- **Unnecessary Turbofish annotations.** The type annotations on the collect function are not needed, as the type can be inferred from the function's return type.

```
// Consumes the Map returning only the vector of the unwrapped data
// If one of the data is still none, then return None
pub fn into_vec_or_none(self) -> Option<Vec<T>> {
    self.data.into_iter().collect::<Option<Vec<_>>>()
}

// Does not consume the map returning only the vector of the unwrapped data
// If one of the data is still none, then return None
pub fn to_refs_or_none(&self) -> Option<Vec<&T>> {
    self.data
        .iter()
        .map(|opt| opt.as_ref())
        .collect::<Option<Vec<_>>>()
}
```

Figure C.8: `threshold-signatures/src/participants.rs#L176–L189`

- **Incomplete docstring for the `get_coefficients` function.**

```
/// Returns the coefficients of the
pub fn get_coefficients(&self) -> Vec<CoefficientCommitment<C>> {
    self.coefficients.to_vec()
}
```

Figure C.9: `threshold-signatures/src/crypto/polynomials.rs#L213–L216`

- **Use of `mut self` on function that does not appear to mutate state.** Consider using `&self` instead.

```
pub(crate) fn verify_proposed_participant_attestation(
    &mut self,
    attestation: &Attestation,
    tls_public_key: PublicKey,
    tee_upgrade_deadline_duration_blocks: u64,
) -> TeeQuoteStatus {
    let expected_report_data = ReportData::V1(ReportDataV1::new(tls_public_key));
    let is_valid = attestation.verify(
        expected_report_data,
        Self::current_time_seconds(),

&self.get_allowed_mpc_docker_image_hashes(tee_upgrade_deadline_duration_blocks),
        &self.allowed_launcher_compose_hashes,
    );

    if is_valid {
        TeeQuoteStatus::Valid
    } else {
        TeeQuoteStatus::Invalid
    }
}
```

Figure C.10: `mpc/crates/contract/src/tee/tee_state.rs#L63–L82`

D. Automated Analysis Tool Configuration

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

Though static analysis tools frequently report false positives, they detect certain categories of issues, such as memory leaks, misspecified format strings, and the use of unsafe APIs, with essentially perfect precision. We recommend periodically running these static analysis tools and reviewing their findings.

We convert tool output to the SARIF file format (e.g., with `clippy-sarif`) when possible, allowing for easy inspection of the results within an IDE (e.g., using VS Code's SARIF Explorer extension).

CodeQL

We used CodeQL to detect known vulnerabilities in the codebase. We installed CodeQL by following CodeQL's installation guide.

```
# Create the Rust database
codeql database create codeql.db --language=rust

# Run all queries for a language
codeql database analyze codeql.db --format=sarif-latest --output=codeql.sarif
```

Figure D.1: The commands used to run CodeQL on the codebase

We also ran internally maintained CodeQL queries on the codebase. For more information about CodeQL, refer to the CodeQL chapter of the Trail of Bits Testing Handbook.

Semgrep

We ran the static analyzer Semgrep with the rulesets shown in figure C.2 to try and identify low-complexity weaknesses in the codebase. To install Semgrep, follow the instructions in Semgrep's documentation.

```
semgrep --metrics=off --sarif --config p/trailofbits
semgrep --metrics=off --sarif --config p/security-audit
```

Figure D.2: The command and rulesets used to run Semgrep on the codebase

zizmor

`zizmor` is a static analysis tool for GitHub Actions. It can be installed using `cargo install --locked zizmor`. We ran `zizmor` with the following command:

```
zizmor --persona pedantic . --format sarif > zizmor_pedantic.sarif
```

Figure D.3: Command used to run `zizmor` and output to SARIF

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 the tool with the pedantic ruleset.

Clippy results from the regular set of rules should always be fixed.

```
# run clippy in regular and pedantic mode and output to SARIF
cargo clippy --message-format=json | clippy-sarif > clippy.sarif
cargo clippy --message-format=json -- -W clippy::pedantic -Wclippy::or_fun_call |
clippy-sarif > clippy_pedantic.sarif
```

Figure D.4: The invocation commands used to run Clippy on the codebase

Dylint

Dylint is a tool for running Rust lints from dynamic libraries, similar to Clippy. We ran the general and supplementary rulesets against the codebase.

```
cargo dylint --no-deps --git https://github.com/trailofbits/dylint --pattern
examples/general -- --message-format=json | clippy-sarif > general.sarif

cargo dylint --no-deps --git https://github.com/trailofbits/dylint --pattern
examples/supplementary -- --message-format=json | clippy-sarif > supplementary.sarif
```

Figure D.5: The command used to run Dylint on the project

test-fuzz

The `test-fuzz` Cargo plugin was used to quickly generate a fuzzing corpus and to implement a fuzzing harness for the safe serialization routines.

To install it, run the following command:

```
cargo install cargo-test-fuzz cargo-afl
```

Figure D.6: The invocations used to install `cargo-test-fuzz` and its dependencies

cargo-edit

`cargo-edit` allows developers to quickly find outdated Rust crates. The tool can be installed with the `cargo install cargo-edit` command, and the `cargo upgrade --incompatible --dry-run` command can be used to find outdated crates.

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 plugin, run the `cargo llvm-cov` command in the crate under test.

cargo-mutants

The `cargo-mutants` Cargo plugin repeatedly mutates the codebase and then runs the test suite to identify under-tested code paths in the codebase. The plugin can be installed via the command `cargo install cargo-mutants`. To run the plugin, simply invoke `cargo mutants` in the root directory of the repository.

E. Kani Proof for the Participant::scalar Function

During our review, we wrote the following Kani proof to verify that the `Participant::scalar` function never returns the zero element. The proof can be checked by running `cargo kani`.

```rust
#[cfg(kani)]
#[kani::proof]
fn check_scalar_zero() {
    use frost_core::{Field, Group};
    use frost_secp256k1::Secp256K1Sha256;
    let zero = <<Secp256K1Sha256 as frost_core::Ciphersuite>::Group as
Group>::Field::zero();

    let x: u32 = kani::any();
    let s = Participant::from(x).scalar::<Secp256K1Sha256>();

    assert_ne!(s, zero);
}
```

Figure E.1: Kani proof that checks that the result of the `scalar` function is never zero

F. Fix Review Results

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

From December 18 to December 19, 2025, Trail of Bits reviewed the fixes and mitigations implemented by the NEAR One team for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue.

In summary, of the 15 issues described in this report, NEAR One has resolved 13 issues, has partially resolved one issue, and has not resolved the remaining issue. The partially resolved issue is related to the security of GitHub actions of the `nearcore` repository, and the unresolved issue is an informational-severity issue that describes certain limitations in the CKD design. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Severity	Status
1	CKD protocol allows a malicious coordinator to control the derived key	High	Resolved
2	CKD response handler lacks access controls	High	Resolved
3	The index function panics when the participant is not in the list	Medium	Resolved
4	AppId Borsh serialization function truncates the AppId bytes if the length is larger than u32::MAX	Informational	Resolved
5	The CKD coordinator hangs forever waiting for all participants	Informational	Resolved
6	Polynomial generation function can panic or create a constant polynomial when called with a large degree	Low	Resolved
7	Reliable broadcast indexes vectors to received indices	Medium	Resolved

8	Resharing with a new participant and threshold equal to one will always fail	Medium	Resolved
9	PolynomialCommitment deserialization function does not trim the coefficient list	Informational	Resolved
10	CKDRequestStorage::get treats any broadcast receive error as fatal	Informational	Resolved
11	Potential credential persistence in artifacts	Informational	Resolved
12	Unpinned external GitHub CI/CD action versions	Low	Resolved
13	Inconsistent handling of threshold in key refresh protocol	Informational	Resolved
14	Unpinned versions and potential credential persistence in nearcore GitHub workflows	Low	Partially Resolved
15	CKD protocol relies solely on MPC node secrets	Informational	Unresolved

Detailed Fix Review Results

TOB-NEARCKD-1: CKD protocol allows a malicious coordinator to control the derived key

Resolved in PR #101, PR #115, and PR #1239. A BLS signature is now used to derive the key. The MPC network computes the BLS signature of `MPC`$_{PK}$`||app_id`, which is also ElGamal encrypted. When the application receives the ElGamal ciphertext, it decrypts it and verifies that the signature is valid. With this protocol, as long as the application always verifies the signature, and the public key associated with the signature is not controlled by a malicious actor, it is not possible for a coordinator to control the derived key, as it must be the signature of a known message. We have suggested a more robust API that returns the derived key only when the signature is valid, preventing API misuse by foregoing the signature verification step.

TOB-NEARCKD-2: CKD response handler lacks access controls

Resolved in PR #1313. Access controls using the `assert_caller_is_attested_participant_and_protocol_active` call were added in the `respond_ckd` function.

TOB-NEARCKD-3: The index function panics when the participant is not in the list

Resolved in PR #117. The `index` function was changed to return a `Result`, which is propagated with the question mark operator.

TOB-NEARCKD-4: AppId Borsh serialization function truncates the AppId bytes if the length is larger than u32::MAX
Resolved in PR #121. `u32::try_from` is now used in the AppId Borsh serialization function.

TOB-NEARCKD-5: The CKD coordinator hangs forever waiting for all participants
Resolved in PR #208. Documentation was added to inform users that callers of the `threshold-signatures` library need to properly manage timeouts.

TOB-NEARCKD-6: Polynomial generation function can panic or create a constant polynomial when called with a large degree
Resolved in PR #163 and PR #206. Checks that prevent potential panics with extremely large degrees were added, and checked arithmetic operations are now used to compute degrees from thresholds.

TOB-NEARCKD-7: Reliable broadcast indexes vectors to received indices
Resolved in PR #164. Out-of-bounds indices are now ignored.

TOB-NEARCKD-8: Resharing with a new participant and threshold equal to one will always fail
Resolved in PR #163 and PR #220. Threshold validation was added to the key generation and key resharing protocols, and checked arithmetic is now used.

TOB-NEARCKD-9: PolynomialCommitment deserialization function does not trim the coefficient list
Resolved in PR #205. The `PolynomialCommitment::new` function is now used instead of custom validation logic.

TOB-NEARCKD-10: CKDRequestStorage::get treats any broadcast receive error as fatal
Resolved in PR #1374. Lagged messages are now handled as non-fatal.

TOB-NEARCKD-11: Potential credential persistence in artifacts
Resolved in PR #198 and PR #1365. `persist-credentials: false` was added to checkout actions.

TOB-NEARCKD-12: Unpinned external GitHub CI/CD action versions
Resolved in PR #198 and PR #1365. Versions are now pinned to the CI/CD actions.

TOB-NEARCKD-13: Inconsistent handling of threshold in key refresh protocol
Resolved in PR #223. A documentation notice was added to explain the requirement between the old and new thresholds during key reshare.

TOB-NEARCKD-14: Unpinned versions and potential credential persistence in nearcore GitHub workflows
Partially resolved in PR #14523. `persist-credentials: false` was added to checkout actions. However, no versions have been pinned to hashes, and `zizmor` reports other potential issues that have not been resolved.

TOB-NEARCKD-15: CKD protocol relies solely on MPC node secrets
Unresolved. The client provided the following context for this decision:

> *This is a feature needed by some of the use cases we envision.*

G. Fix Review Status Categories

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

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

About Trail of Bits

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

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

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

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

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

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

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

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

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

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