CODESPECT

# Typhoon Mixer

## SECURITY ASSESSMENT REPORT

25 May, 2025

*Prepared for Typhoon*

# Contents

# 1 About CODESPECT

CODESPECT is a specialized smart contract security firm dedicated to ensure the safety, reliability, and success of blockchain projects. Our services include comprehensive smart contract audits, secure design and architecture consultancy, and smart contract development across leading blockchain platforms such as Ethereum (Solidity), Starknet (Cairo), and Solana (Rust).

At CODESPECT, we are committed to build secure, resilient blockchain infrastructures. We provide strategic guidance and technical expertise, working closely with our partners from concept development through deployment. Our team consists of blockchain security experts and seasoned engineers who apply the latest auditing and security methodologies to help prevent exploits and vulnerabilities in your smart contracts.

**Smart Contract Auditing:** Security is at the core of everything we do at CODESPECT. Our auditors conduct thorough security assessments of smart contracts written in Solidity, Cairo, and Rust, ensuring that they function as intended without vulnerabilities. We specialize in providing tailored security solutions for projects on EVM-compatible chains and Starknet. Our audit process is highly collaborative, keeping clients involved every step of the way to ensure transparency and security. Our team is also dedicated to cutting-edge research, ensuring that we stay ahead of emerging threats.

**Secure Design & Architecture Consultancy:** At CODESPECT, we believe that secure development begins at the design phase. Our consultancy services offer deep insights into secure smart contract architecture and blockchain system design, helping you build robust, secure, and scalable decentralized applications. Whether you're working with Ethereum, Starknet, or other blockchain platforms, our team helps you navigate the complexity of blockchain development with confidence.

**Tailored Cybersecurity Solutions**: CODESPECT offers specialized cybersecurity solutions designed to minimize risks associated with traditional attack vectors, such as phishing, social engineering, and Web2 vulnerabilities. Our solutions are crafted to address the unique security needs of blockchain-based applications, reducing exposure to attacks and ensuring that all aspects of the system are fortified.

With a focus on the intersection of security and innovation, CODESPECT strives to be a trusted partner for blockchain projects at every stage of development and for each aspect of security.

# 2 Disclaimer

**Limitations of this Audit:** This report is based solely on the materials and documentation provided to CODESPECT for the specific purpose of conducting the security review outlined in the Summary of Audit and Files. The findings presented in this report may not be comprehensive and may not identify all possible vulnerabilities. CODESPECT provides this review and report on an "as-is" and "as-available" basis. You acknowledge that your use of this report, including any associated services, products, protocols, platforms, content, and materials, is entirely at your own risk.

**Inherent Risks of Blockchain Technology:** Blockchain technology is still evolving and is inherently subject to unknown risks and vulnerabilities. This review focuses exclusively on the smart contract code provided and does not cover the compiler layer, underlying programming language elements beyond the reviewed code, or any other potential security risks that may exist outside of the code itself.

**Purpose and Reliance of this Report:** This report should not be viewed as an endorsement of any specific project or team, nor does it guarantee the absolute security of the audited smart contracts. Third parties should not rely on this report for any purpose, including making decisions related to investments or purchases.

**Liability Disclaimer:** To the maximum extent permitted by law, CODESPECT disclaims all liability for the contents of this report and any related services or products that arise from your use of it. This includes but is not limited to, implied warranties of merchantability, fitness for a particular purpose, and non-infringement.

**Third-Party Products and Services:** CODESPECT does not warrant, endorse, or assume responsibility for any third-party products or services mentioned in this report, including any open-source or third-party software, code, libraries, materials, or information that may be linked to, referenced by, or accessible through this report. CODESPECT is not responsible for monitoring any transactions between you and third-party providers. We strongly recommend conducting thorough due diligence and exercising caution when engaging with third-party products or services, just as you would for any other product or service transaction.

**Further Recommendations:** We advise clients to schedule a re-audit after any significant changes to the codebase to ensure ongoing security and reduce the risk of newly introduced vulnerabilities. Additionally, we recommend implementing a bug bounty program to incentivize external developers and security researchers to identify and disclose potential vulnerabilities safely and responsibly.

**Disclaimer of Advice:** FOR AVOIDANCE OF DOUBT, THIS REPORT, ITS CONTENT, AND ANY ASSOCIATED SERVICES OR MATERIALS SHOULD NOT BE CONSIDERED OR RELIED UPON AS FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER PROFESSIONAL ADVICE.

# 3 Risk Classification

| Severity Level | Impact: High | Impact: Medium | Impact: Low |
|---|---|---|---|
| **Likelihood: High** | Critical | High | Medium |
| **Likelihood: Medium** | High | Medium | Low |
| **Likelihood: Low** | Medium | Low | Low |

Table 1: Risk Classification Matrix based on Likelihood and Impact

### 3.1 Impact

- **High** - Results in a substantial loss of assets (more than 10%) within the protocol or causes significant disruption to the majority of users.
- **Medium** - Losses affect less than 10% globally or impact only a portion of users, but are still considered unacceptable.
- **Low** - Losses may be inconvenient but are manageable, typically involving issues like griefing attacks that can be easily resolved or minor inefficiencies such as gas costs.

### 3.2 Likelihood

- **High** - Very likely to occur, either easy to exploit or difficult but highly incentivized.
- **Medium** - Likely only under certain conditions or moderately incentivized.
- **Low** - Unlikely unless specific conditions are met, or there is little-to-no incentive for exploitation.

### 3.3 Action Required for Severity Levels

- **Critical** - Must be addressed immediately if already deployed.
- **High** - Must be resolved before deployment (or urgently if already deployed).
- **Medium** - It is recommended to fix.
- **Low** - Can be fixed if desired but is not crucial.

In addition to High, Medium, and Low severity levels, CODESPECT utilizes two other categories for findings: **Informational** and **Best Practices**.

a) **Informational** findings do not pose a direct security risk but provide useful information the audit team wants to communicate formally.

b) **Best Practices** findings indicate that certain portions of the code deviate from established smart contract development standards.

# 4 Executive Summary

This document presents the results of a security assessment conducted by CODESPECT for Typhoon. Typhoon is a privacy-focused protocol designed for confidential fund transfers, inspired by the well-known Tornado Cash.

The scope of this audit includes the Cairo-based Typhoon contracts, specifically the factory responsible for creating private transfer pools. These pools enable anonymous transfers between accounts, serving as a core component of the protocol's privacy mechanism.

**The audit was performed using:**

a) Manual analysis of the codebase.

b) Dynamic analysis of programs, execution testing.

CODESPECT found nine points of attention, one classified as `Critical`, three classified as `High`, three classified as `Medium`, one classified as `Info` and one classified as `Best Practice`. All of the issues are summarised in Table 2.

**Audit Conclusion**

CODESPECT conducted a thorough review of the contracts within the agreed scope. However, due to insufficient testing and documentation, along with the high number of critical and high issues identified, we strongly recommend a follow-up audit in the near future. Additionally, one of the protocol's core storage structures was significantly changed — replacing the previous Merkle tree with a lazy tower implementation as part of the fixes. This major architectural update warrants further testing and external review to ensure robustness and correctness.

**Organisation of the document is as follows:**

- **Section 5** summarizes the audit.
- **Section 6** describes the system overview.
- **Section 7** presents the issues.
- **Section 8** contains additional notes for the audit.
- **Section 9** discusses the documentation provided by the client for this audit.
- **Section 10** presents the compilation and tests.

## Issues found:

| Severity | Unresolved | Fixed | Acknowledged |
|---|---|---|---|
| Critical | 0 | 1 | 0 |
| High | 0 | 3 | 0 |
| Medium | 0 | 3 | 0 |
| Informational | 0 | 1 | 0 |
| Best Practices | 0 | 1 | 0 |
| **Total** | **0** | **9** | **0** |

Table 2: Summary of Unresolved, Fixed, and Acknowledged Issues

# 5 Audit Summary

| | |
|---|---|
| **Audit Type** | Security Review |
| **Project Name** | Typhoon Mixer |
| **Type of Project** | Cryptocurrency Mixer |
| **Duration of Engagement** | 3 Days |
| **Duration of Fix Review Phase** | 2 Days |
| **Draft Report** | May 9, 2025 |
| **Final Report** | May 25, 2025 |
| **Repository** | typhoon-contracts |
| **Commit (Audit)** | a148cf0c83d28e98b1bd21af301bfd55f9a9c3f7 |
| **Commit (Final)** | 3b54a2b1bd9680fcb807584d137833bb7995174a |
| **Documentation Assessment** | Low |
| **Test Suite Assessment** | Not evaluated |
| **Auditors** | Talfao, Kalogerone |

Table 3: Summary of the Audit

## 5.1 Scope - Audited Files

| | Contract | LoC |
|---|---|---|
| 1 | NoteAccount.cairo | 73 |
| 2 | lib.cairo | 15 |
| 3 | Pool.cairo | 323 |
| 4 | Typhoon.cairo | 143 |
| 5 | Hasher.cairo | 80 |
| | **Total** | **634** |

**Scope information**

The functionality related to the rewarding mechanism was not reviewed in the initial phase of the audit, as the Typhoon team indicated that this feature would be removed in the final deployed version.

## 5.2 Findings Overview

| | Finding | Severity | Update |
|---|---|---|---|
| 1 | Encrypted notes can be arbitrarily altered and signatures can be replayed | Critical | Fixed |
| 2 | Merkle tree overwrite issue after max depth reached | High | Fixed |
| 3 | `withdraw_fee` remains stuck in contract with no withdrawal mechanism | High | Fixed |
| 4 | newRootIndex should not use modulo with `ROOT_HISTORY_SIZE` | High | Fixed |
| 5 | Inconsistent `notesCount` handling results in inconsistent results in multiple functions | Medium | Fixed |
| 6 | Updating the `current_day` always adds an additional day | Medium | Fixed |
| 7 | `notesCount` is not updated during the execution of `updateNotes(...)` | Medium | Fixed |
| 8 | Adding a pool with the same token and denomination of another pool will override it in the `pools` mapping | Info | Fixed |
| 9 | Skip relayer transfer if address is not properly set | Best Practices | Fixed |

# 6 System Overview

The Typhoon protocol is composed of several modular contracts that work together to enable privacy-preserving fund transfers.

The central component is the `Typhoon` contract, which acts as an interface between users and the underlying pool logic. It handles deposits and withdrawals and also serves as a factory for creating new `Pool` contracts. Each `Pool` is deployed using a predefined class hash that is stored in the `Typhoon` contract upon deployment and remains immutable thereafter.

A `Pool` contract is responsible for holding user funds, which can later be withdrawn privately. Each pool is specific to a single token and a fixed denomination, defined by the `denomination` storage variable set during deployment. Users deposit funds via the `processDeposit(...)` function, which must be called through the `Typhoon` contract:

```
fn processDeposit(ref self: ContractState, _from: ContractAddress, reward: bool, commitment: u256) -> (u256,
↪  Array<u256>)
```

Each deposit is uniquely associated with a `commitment`, preventing duplicate deposits linked to one commitment. During the deposit, funds are transferred from the user to the pool, and the commitment is hashed along with the current day to generate a Merkle tree leaf. A new Merkle root is computed and stored in the `roots` mapping under a new index.

Withdrawals require a zero-knowledge proof generated off-chain during the creation of the encrypted note at the time of deposit. The proof is verified by the `Verifier` contract, which, upon successful validation, returns the public inputs to the `Pool` contract. These include:

- `root` – used to verify that the deposit is part of a valid Merkle tree.
- `nullifierHash` – checked to prevent double-spending of the same deposit.
- `recipient` – the address that will receive the withdrawn funds.
- `relayer` and `relayerFee` – optional parameters that are not used in the current version.

The `Pool` contract verifies that the provided `root` exists in its known history and that the `nullifierHash` has not already been marked as spent. If all checks pass, the funds are transferred to the `recipient`.

Additionally, the protocol includes a `NoteAccount` contract that stores encrypted notes on-chain. These notes represent the encrypted metadata of deposits and are essential for generating valid withdrawal proofs. Notes can be added or updated via the following functions:

```
fn addNote(ref self: TContractState, pubKey: EthAddress, encryptedNote: Span<u256>, msg_hash: u256, r: u256, s: u256, v:
↪  u32);

fn updateNotes(ref self: TContractState, pubKey: EthAddress, msg_hash: u256, r: u256, s: u256, v: u32, newNotes:
↪  Span<Span<u256>>);
```

# 7 Issues

## 7.1 [Critical] Encrypted notes can be arbitrarily altered and signatures can be replayed

**File(s)**: `NoteAccount.cairo`

**Description**: Once the deposit is made, an encrypted note is created inside `NoteAccount.cairo` via an off-chain process—this process then generates a proof based on the information stored in `NoteAccount.cairo`. The encrypted note is added through `addNote(...)`, and notes are updated via `updateNotes(...)`:

```
fn addNote(ref self: ContractState, pubKey: EthAddress, encryptedNote: Span<u256>, msg_hash: u256, r: u256, s: u256, v:
↪  u32);

fn updateNotes(ref self: ContractState, pubKey: EthAddress, msg_hash: u256, r: u256, s: u256, v: u32, newNotes:
↪  Span<Span<u256>>)
```

The main issue is that the signature can be replayed. There is no mechanism to ensure freshness in the signed data. In general, a nonce should be included to make every signed message unique.

There is also a secondary minor issue in the implementation of these functions: `msg_hash` is passed as an input parameter without enforcing any link to the actual `encryptedNote` or `newNotes`. This means the contract does not verify that the message hash corresponds to the provided data. If the owner modifies these values, they would only risk their own funds—assuming the primary replay issue is resolved.

**Impact**: Arbitrary alteration of encrypted notes and potential theft of funds.

**Recommendation(s)**: Prevent replay attacks by introducing a nonce.

**Status**: Fixed

**Update from Typhoon**: Solved 1fa6a6d6462c66d8ce271b077a0420a7bcbfafdf

## 7.2 [High] Merkle tree overwrite issue after max depth reached

**File(s)**: `Pool.cairo`

**Description**: `Pools` Merkle trees get a maximum number of levels during construction:

```
self.levels.write(10);
```

For Merkle trees, 10 levels mean that there is a maximum of 1024 deposits, but the contract is intended to be used for more deposits. Also, the contract allows for 1024+ deposits to go through without reverting, which can cause serious issues. For the 1025th deposit, the issue is that the code doesn't realize it has exceeded the tree depth.

When it processes index 1024 (1025th deposit):

- It updates subtree[0], overwriting the value from index 0;
- But the rest of the tree path is different;
- This creates a corrupted tree structure where multiple leaves share partial paths;

For a user who deposited at index 0:

- If they try to withdraw after index 1024 has been processed;
- Their Merkle proof will fail because subtree[0] has been overwritten;
- They will be unable to withdraw their funds;

**Impact**: Earlier deposits become unrecoverable once the tree capacity is exceeded, but not in a straightforward `index % 1024` pattern.

**Recommendation(s)**: Potentially set the max level to a higher amount.

**Status**: Fixed

**Update from Typhoon**: Solved 1fa6a6d6462c66d8ce271b077a0420a7bcbfafdf

**Update from CODESPECT**: The issue has been resolved by replacing the previous tree implementation with a lazy tower data structure. This new implementation closely follows the Solidity reference implementation available here, which CODESPECT reviewed and found no immediate vulnerabilities.

However, given that this change affects one of the core components of the protocol, we strongly recommend performing both basic and in-depth functional testing to ensure correctness. At minimum, we suggest writing a test that inserts at least **10 items** — ideally more — to validate the new structure thoroughly.

Additionally, due to the significance of the architectural change and the number of previous issues identified, we advise conducting a **follow-up audit** with another independent security provider for further assurance.

Lastly, we noted that the updated `insert(...)` function includes **unused index-related variables** that should be removed to maintain code clarity and avoid potential confusion.

## 7.3 [High] `withdraw_fee` remains stuck in contract with no withdrawal mechanism

**File(s)**: `Pool.cairo`

**Description**: The `withdraw_fee` is deducted from each withdrawal and represents 0.5% of the denomination amount:

```
IERC20Dispatcher { contract\_address: self.token.read() }
.transfer(
recipient, // @audit-issue withdraw_fee is not sent anywhere
((self.denomination.read() - self.withdraw\_fee.read()) - \*value\[5])
\+ reward,
);
```

While the fee is correctly subtracted, it is not transferred to any address and remains stuck within the contract. The fee is intended to reward liquidity providers. Although the reward mechanism may be removed before launch, the fee mechanism itself is meant to stay. However, there is currently no function available to withdraw or utilize the accumulated fees.

**Impact**: Fees accumulate in the contract with no way to access or distribute them. Since the contract is not upgradeable, this results in a permanent loss of funds.

**Recommendation(s)**: Implement functionality to withdraw the accumulated fees or automatically transfer the fee to a designated address during each withdrawal.

**Status**: Fixed

**Update from Typhoon**: Solved 1fa6a6d6462c66d8ce271b077a0420a7bcbfafdf

**Update from CODESPECT**: The issue was addressed by introducing a new function for withdrawing profits. However, a new function for updating the fee was also added. The problem with this `setWithdrawFee` function is that it does not enforce any upper bound on the fee value, which may lead to potential misuse or misconfiguration.

**Update from Typhoon**: Solved in d893775a7bff25422731b24240fc9d81864c84e9

## 7.4 [High] newRootIndex should not use modulo with `ROOT_HISTORY_SIZE`

**File(s)**: `Pool.cairo`

**Description**: During the deposit process, the commitment is added to the Merkle tree and a new root is generated. This root is stored in the `roots` mapping, with its index tracked by the `newRootIndex` variable. The index is calculated in the `insert(...)` function as follows:

```
let newRootIndex: u32 = (self.current_root_index.read() + 1) % ROOT_HISTORY_SIZE;
```

The issue lies in the use of the modulo operation with `ROOT_HISTORY_SIZE`, which limits the number of stored roots to 30. This could result in overwriting a root that is still needed for a pending withdrawal, potentially causing users to lose access to their funds. Additionally, the `isKnownRoot(...)` function does not check for a root stored at index zero, which could lead to an unrecognised valid root, especially affecting the first deposit.

**Impact**:

* Potential loss of funds due to overwriting active roots. * Failure to recognise the root of the first deposit.

**Recommendation(s)**:Remove the modulo operation with `ROOT_HISTORY_SIZE` to avoid overwriting unclaimed roots.

**Status**: Fixed

**Update from Typhoon**: Solved 1fa6a6d6462c66d8ce271b077a0420a7bcbfafdf

## 7.5 [Medium] Inconsistent `notesCount` handling results in inconsistent results in multiple functions

**File(s)**: `NoteAccount.cairo`

**Description**: In the `addNote(...)` function the first note of a user gets registered at the `notesCount` mapping at index `1` and later at the `notes` mapping it uses the same index as the latest `notesCount` (e.g. `1` for the first note):

```
fn addNote(ref self: ContractState, pubKey: EthAddress, encryptedNote: Span<u256>, msg_hash: u256, r: u256, s: u256, v:
↪  u32) {
    // will panic if the signature is invalid
    verify_signature(pubKey, msg_hash, r, s, v);
    self.notesCount.entry(pubKey).write(self.notesCount.read(pubKey) + 1);
    self.notes.entry(pubKey).entry(self.notesCount.read(pubKey)).write((*encryptedNote[0], *encryptedNote[1],
    ↪  *encryptedNote[2], *encryptedNote[3], *encryptedNote[4], *encryptedNote[5], *encryptedNote[6]));
}
```

However, in other functions like `getNotes(...)` and `eraseNotes(...)` loops start from index `0` and end when `i = notesCount`:

```
fn getNotes(self: @ContractState, pubKey: EthAddress) -> Array<(u256, u256, u256, u256, u256, u256, u256)> {
    let mut notes: Array = ArrayTrait::<(u256, u256, u256, u256, u256, u256, u256)>::new();
    let mut i: u256 = 0;
    if(self.notesCount.read(pubKey) == 0){
        let mut a: Array = ArrayTrait::<(u256, u256, u256, u256, u256, u256, u256)>::new();
        a.append((0,0,0,0,0,0,0));
        return a;
    }
    loop{
        let note = self.notes.entry(pubKey).entry(i).read();
        notes.append(note);
        i+=1;
        if(self.notesCount.read(pubKey) == i){
            break;
        }

   };
   return notes;
}
```

```
fn eraseNotes(ref self: ContractState, pubKey: EthAddress){
    let mut i: u256 = 0;
    loop{
        self.notes.entry(pubKey).entry(i).write((0,0,0,0,0,0,0));
        i+=1;
        if(self.notesCount.read(pubKey) == i){
            break;
        }
    };
}
```

This results on their first iteration being unnecessary since the `notes` mapping is empty at that index. Also, the loop stops 1 iteration before it should, because it doesn't process the `notes` mapping at the `notesCount` index, which stores information. Contradictory to this logic, the `updateNotes(...)` function updates the `notes` mapping starting from index `0`, so `getNotes(...)` and `eraseNotes(...)` work correctly for `notes` that have been created through this function:

```
fn updateNotes(ref self: ContractState,pubKey: EthAddress, msg_hash: u256, r: u256, s: u256, v: u32  ,newNotes:
↪  Span<Span<u256>>){
      verify_signature(pubKey, msg_hash, r, s, v);
      let mut i: u32 = 0;
      eraseNotes(ref self,pubKey);
      loop{
          self.notes.entry(pubKey).entry(i.into()).write((*newNotes[i][0], *newNotes[i][1], *newNotes[i][2],
          ↪  *newNotes[i][3], *newNotes[i][4], *newNotes[i][5], *newNotes[i][6]));
          i+=1;
          if(newNotes.len() == i){
              break;
          }
     }
}
```

**Impact**: The `getNotes(...)` and `eraseNotes(...)` will not work correctly for notes that have created using the `addNotes(...)` function. The `updateNotes(...)` function will also not work correctly since it uses the `eraseNotes(...)` function itself.

**Recommendation(s)**: Choose a consistent starting index and use it for all the loops.

**Status**: Fixed

**Update from Typhoon**: Solved 1fa6a6d6462c66d8ce271b077a0420a7bcbfafdf

## 7.6 [Medium] Updating the `current_day` always adds an additional day

**File(s)**: `Pool.cairo`

**Description**: Deposits and withdrawals always call the `updateDay(...)` function to check if a day has passed since the storage variable `current_day`. However, if at least a day has passed and `current_day` needs to be updated, an additional day is always added:

```
fn updateDay(ref self: ContractState) {
    let mut cur_day = self.current_day.read();
    let mut dif: u256 = 0;
    if cur_day < get_block_timestamp().into() {
        dif = get_block_timestamp().into() - cur_day;
        if dif > day.into() {
            let days = getDaysPassed(@cur_day);
            self.current_day.write(cur_day + (day.into() * days) + day.into()); // @audit-issue an additional day gets
            ↪  added
        }
    }
}
```

**Impact**: Inaccurate tracking of the days, since every time the `current_day` gets updated, an additional day gets added.

**Recommendation(s)**: Remove the `+ day.into()` part of the calculation.

**Status**: Fixed

**Update from Typhoon**: Solved 1fa6a6d6462c66d8ce271b077a0420a7bcbfafdf

## 7.7 [Medium] `notesCount` is not updated during the execution of `updateNotes(...)`

**File(s)**: `NoteAccount.cairo`

**Description**: The `notesCount` storage mapping tracks the number of notes associated with each public key.

However, this value is not updated during the execution of `updateNotes(...)`. As a result, after a note update, the stored count may no longer reflect the actual number of notes. The internal `eraseNotes(...)` function should reset the note count to zero, and during the loop in `updateNotes(...)`, the count should be incremented on each iteration.

```
loop {
    self.notes.entry(pubKey).entry(i.into()).write((
        *newNotes[i][0], *newNotes[i][1], *newNotes[i][2],
        *newNotes[i][3], *newNotes[i][4], *newNotes[i][5],
        *newNotes[i][6]
    ));
    i += 1;
    if (newNotes.len() == i) {
        break;
    }
    // @audit update the note count
}
```

**Impact**: Incorrect tracking of note counts can lead to inconsistencies and unexpected behavior in contract logic that depends on the accurate number of notes per public key.

**Recommendation(s)**: Ensure `eraseNotes(...)` resets the count to zero, and increment `notesCount` in each iteration of the update loop to maintain consistency.

**Status**: Fixed

**Update from Typhoon**: Solved 1fa6a6d6462c66d8ce271b077a0420a7bcbfafdf

## 7.8 [Info] Adding a pool with the same token and denomination of another pool will override it in the `pools` mapping

**File(s)**: `Typhoon.cairo`

**Description**: The `pools` mapping keeps track of all the pools deployed according to their token and denomination only. If a new pool gets deployed with the same token and denomination of another pool, the new pool address will override the old one's at the mapping, which is only used in the `getPool(...)` function.

```
fn addPool(
    ref self: ContractState, _token: ContractAddress, _denomination: u256, _day: u256,
) {
    ...
    self.pools.entry(_token).entry(_denomination).write(pool_address);
    self.pool_salt.write(self.pool_salt.read() + 1);
    self.allowed_pools.entry(pool_address).write(true);
}
```

**Status**: Fixed

**Update from Typhoon**: Solved 1fa6a6d6462c66d8ce271b077a0420a7bcbfafdf

## 7.9 [Best Practice] Skip relayer transfer if address is not properly set

**File(s)**: `Pool.cairo`

**Description**: The withdrawal process may involve a relayer that receives a fee. Currently, this mechanism is inactive, and `value[5]` should always be zero, as defined by the Typhoon team. However, this assumption may change in the future.

```
if (*value[5] > 0) {
    IERC20Dispatcher { contract_address: self.token.read() }
        .transfer(relayer, *value[5]);
}
```

The current check only verifies if `value[5] > 0`, but it does not confirm whether the `relayer` address is properly set. A better practice would be to additionally check that `relayer` is not the zero address, ensuring it was correctly set in the note data.

**Impact**: Potential loss of relayer fee if the `relayer` address is unset or incorrectly set in the encrypted note.

**Recommendation(s)**: Add a check to ensure that the `relayer` address is not the zero address before executing the transfer.

**Status**: Fixed

**Update from Typhoon**: Solved 1fa6a6d6462c66d8ce271b077a0420a7bcbfafdf

**Update from CODESPECT**: The current fix introduces a logic error—the condition was implemented incorrectly. As it stands, the condition only passes when the `relayer` is *not* set. Specifically, the check `relayer == contract_address_const::<0>()` should be updated to `value[5] > 0 && relayer != contract_address_const::<0>()` to ensure correct behavior.

**Update from Typhoon**: Solved 3b54a2b1bd9680fcb807584d137833bb7995174a

# 8 Additional Notes

This section provides supplementary auditor observations regarding the code. These points were not identified as individual issues but serve as informative recommendations to enhance the overall quality and maintainability of the codebase.

- The `day` constant should be of type `u64`, as timestamps are typically stored using this type.
- The `day` constant should be named in uppercase (e.g., `DAY`) to follow naming conventions for constants.
- The `constructor` in `Typhoon.cairo` can use `ClassHash` instead of `felt252` for the `_poolClassHash` parameter.
- Avoid using magic numbers for BIPS in percentage calculations—use a named constant instead (e.g., `const BIPS = 100`).
- In the `onepercent` calculation, multiplication by 1 is unnecessary and can be omitted.
- In the `withdraw_fee` calculation, multiplication by 50 and later division by 100 is unnecessary and can be simplified to a single division by 2.
- During the execution of the `withdraw(...)` function in `Typhoon.cairo`, it would be beneficial to verify whether the specified `pool` is allowed.

**Update from Typhoon:** Solved 1fa6a6d6462c66d8ce271b077a0420a7bcbfafdf

# 9 Evaluation of Provided Documentation

The **Typhoon** documentation was primarily provided through **NatSpec comments**. These comments explain the purpose of each function, its parameters, and the return values. In addition, some in-line comments were included to clarify specific sections of the code. However, certain parts of the code lacked such comments, which would have improved the overall readability and understanding.

The documentation provided by **Typhoon** ensures a basic understanding of the protocol's core functionality. Nonetheless, the availability of comprehensive public technical documentation and more detailed natspec comments would significantly enhance the clarity and accessibility of the protocol for external reviewers and contributors.

Throughout the evaluation process, the **Typhoon** team was consistently available and highly responsive, promptly addressing all questions raised by **CODESPECT**.

# 10 Test Suite Evaluation

The test suite could not be properly evaluated, as the audited version of the protocol contained syntax errors that prevented the tests from running. As a result, the current test suite appears to be incomplete and will need to be rebuilt from the ground up.

A robust test suite should comprehensively cover both standard and edge-case scenarios. This includes all basic flows—such as deposits and withdrawals—as well as more complex interactions between contracts. Additionally, the protocol team is encouraged to define key invariants that the system must uphold and to construct tests that verify these invariants under various conditions.