Chainflip Protocol

Security Assessment

July 17, 2023

Prepared for:

Martin Rieke

Chainflip

Prepared by: **Nat Chin**, **Fredrik Dahlgren**, **Guillermo Larregay**, **Joop van de Pol**, **Will Song**, and **Kurt Willis**

About Trail of Bits

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

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

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

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

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

To keep up to date with our latest news and announcements, please follow @trailofbits on Twitter 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

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be 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 the controls and security properties of software. 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. Their use is also limited by the time and resource constraints of a project.

Table of Contents

About Trail of Bits 1

Notices and Remarks 2

Table of Contents 3

Executive Summary 5

Project Summary 8

Project Targets 9

Project Goals 10

Project Coverage 13

Automated Testing 17

Codebase Maturity Evaluation 22

Summary of Findings 27

Detailed Findings 29

1. Step 2 of the handover protocol can be forged 29
2. Hash function is used as KDF in handover protocol 31
3. Ceremony participants can send many delayed messages 32
4. Binding value can be zero 33
5. The Chainflip back end and smart contracts have vulnerable dependencies 35
6. Potential panic in KeyId::from_bytes 38
7. Solidity compiler optimizations can be problematic 39
8. ERC-20 token transfer fails for certain tokens 40
9. addGasNative is missing check for nonzero value 42
10. StakeManager contains unnecessary receive function 43
11. Missing events for important operations 44
12. Nonstandard ERC-20 tokens get stuck when depositing 46
13. transfer can fail due to a fixed gas stipend 48
14. Low number of block confirmations configured for external blockchains 49
15. Hard to diagnose error from default behavior during signer nomination 51
16. Failed broadcast nominees are not punished if epoch ends during broadcast 53
17. Nominated broadcast signer does not always report failures in engine 55
18. Threshold signature liveness protection does not account for previously punished validators 57
19. A malicious minority can ruin liveness 58
20. Validators can report nonparticipants in ceremonies 60

21. Staker funds can be locked via front-running 62
22. Unbounded loop execution may result in out-of-gas errors 64
23. Anyone can cause the Chainflip engine to panic 67
24. Failed deposits are incorrectly witnessed as having succeeded 69
25. Validators are not reimbursed for transactions submitted to external chains 73
26. MEV incentives are unclear and require further investigation 74
A. Vulnerability Categories 78
B. Code Maturity Categories 80
C. Automated Testing 82
D. Code Quality Recommendations 90
E. Risks of Third-Party Contract Calls 92
F. Incident Response Recommendations 93
G. Fix Review Results 95

Executive Summary

Engagement Overview

Chainflip engaged Trail of Bits to review the security of the Chainflip protocol. Chainflip implements a decentralized cross-chain swapping protocol based on Substrate. The protocol supports swaps to and from Ethereum, Bitcoin, and Polkadot.

A team of six consultants conducted the review from March 6 to April 28, 2023, for a total of 24 engineer-weeks of effort. The engagement was divided into two parts. The first six engineer-weeks covered the implementation of the FROST threshold signature scheme used to sign outgoing transactions. The remaining 18 engineer-weeks comprised a security review of the rest of the system. During the engagement we performed static and dynamic testing of the codebase, using both automated and manual processes. We had full access to both source code and documentation.

Observations and Impact

Overall, we found that the Chainflip code is defensively written and easy to read. The project is divided into well-defined components with limited responsibilities, which makes the code easier to review and maintain and provides a good foundation for the project going forward. However, the system still suffers from a high aggregate complexity that makes it difficult for a single developer or reviewer to understand the security properties of all the different components in the system. We also found that the State Chain and engine both lack automated testing such as property testing or fuzz testing. This is particularly relevant for the automated market maker (AMM) where any issues related to arithmetic operations may lead to significant monetary repercussions.

Recommendations

Based on the codebase maturity evaluation and findings identified during the security review, Trail of Bits recommends that Chainflip take the following steps:

- **Remediate the findings disclosed in this report.** The findings described in the Detailed Findings section collectively pose a severe risk to the Chainflip protocol. These findings should be addressed as part of a direct remediation or as part of any refactor that may occur when addressing other recommendations.
- **Reduce the overall complexity of the system.** The interactions between system components that are triggered by a single user operation are often very complex and span many varied technology stacks with different security assumptions and guarantees. This makes it difficult for any one developer to understand the security properties provided by the different components. We recommend that Chainflip reduce the system's overall complexity, including reducing the number of

interactions between different components and simplifying the deposit flow on Ethereum.

- **Implement automated testing.** The Ethereum smart contracts implement several Echidna tests, and the Chainflip back end implements unit and integration tests using `cargo test`. However, there is no property testing or fuzz testing implemented for the back end. Many back-end components would benefit from property testing using a framework such as `proptest`. In particular, the AMM implementation should implement property testing to exercise the implementation on unexpected and invalid inputs. We also recommend that Chainflip consider implementing differential fuzzing of the AMM against Uniswap v3.

- **Review interactions between the State Chain and external chains.** Many of the issues we identified during the engagement are related to interactions between the State Chain and external chains. We recommend that Chainflip review and document these interactions to ensure that the happy path and any edge cases are well understood.

- **Ensure that the documentation does not diverge from the implementation.** We found several instances, both in the white paper and in README files in the repository, where the documentation differed from the actual implementation. We recommend that Chainflip regularly review the documentation to ensure that it matches the implementation. This makes it easier to onboard new users and developers, identify areas of improvement, and ensure that various security properties are upheld.

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

EXPOSURE ANALYSIS

Severity	*Count*
High	4
Medium	5
Low	3
Informational	11
Undetermined	3

CATEGORY BREAKDOWN

Category	*Count*
Configuration	2
Cryptography	2
Data Validation	14
Denial of Service	3
Error Reporting	2
Patching	1
Undefined Behavior	2

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager
dan@trailofbits.com

Anne Marie Barry, Project Manager
annemarie.barry@trailofbits.com

The following engineers were associated with this project:

Nat Chin, Consultant
nat.chin@trailofbits.com

Fredrik Dahlgren, Consultant
fredrik.dahlgren@trailofbits.com

Guillermo Larregay, Consultant
guillermo.larregay@trailofbits.com

Joop van de Pol, Consultant
joop.vandepol@trailofbits.com

Will Song, Consultant
will.song@trailofbits.com

Kurt Willis, Consultant
kurt.willis@trailofbits.com

Project Timeline

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

Date	Event
February 16, 2023	Pre-project kickoff call for the cryptography review
March 10, 2023	Status update meeting #1
March 17, 2023	Status update meeting #2
March 22, 2023	Kickoff call for the secure code review
April 11, 2023	Status update meeting #3
April 14, 2023	Status update meeting #4
April 21, 2023	Status update meeting #5
April 28, 2023	Delivery of report draft
April 28, 2023	Report readout meeting
July 17, 2023	Delivery of final report

Project Targets

The engagement involved a review and testing of the targets listed below.

Chainflip back end

Repository	chainflip-io/chainflip-backend
Version (Cryptography review)	`887d55d40e9eca0ab83d2c4f6c95abc3c7094289`
Versions (Secure code review)	`03dc9c0378675c5dff80b2fb59e01c7076383b3a`
	`d172cad95456061027e363491bc6771cb4b534c6`
Type	Rust
Platform	Linux

Chainflip Ethereum contracts

Repository	chainflip-io/chainflip-eth-contracts
Version (Cryptography review)	`55402af70f6f68636b4bac424bd75c1b90d339fd`
Versions (Secure code review)	`6f43be3103de2486aae55d2e5a170d40e69baece`
	`4ed848684aa07be818da85923ca701e0e36ef602`
	`599a6a6618a750c059d8b6258aff347dbea3970e`
Type	Solidity
Platform	Ethereum

Project Goals

The engagement was scoped to provide a security assessment of the Chainflip protocol. The project was divided into two separate parts: a cryptographic review of the FROST threshold signature scheme implementation, and a security review of the Chainflip back end and smart contracts.

During the cryptographic review of the FROST threshold signature scheme, we sought to answer the following non-exhaustive list of questions:

- Does the implementation follow the protocol specified in the FROST paper?
- Are values such as polynomial commitments and curve points received from other nodes validated correctly?
- Is it possible for a node to send different polynomial commitments to different nodes?
- Does the implementation securely generate random scalars?
- Are secret scalars zeroized as they go out of scope?
- Are Schnorr proofs of possession generated and verified correctly?
- Are key shares and the shared public key verified against the corresponding commitments?
- Are presignatures securely deleted as they are used?
- Is the final signature verified by each node?
- Can malicious nodes cause honest nodes to be slashed during the blame stage?
- Does the implementation protect against cross-session replay attacks?
- Is it possible to trick a node into accepting invalid messages? Or can a node impersonate another node?
- Is the unit and integration test coverage good enough for each component of the implementation?
- Does each component test for adversarial behavior?
- Is key resharing correctly implemented?

During the security review of the Chainflip back end and smart contracts, we investigated the following non-exhaustive list of questions:

- Do all extrinsics implement proper access controls?
- Are errors and panics in extrinsics handled correctly? Is storage rolled back when an error occurs?
- Can users launch denial-of-service (DoS) attacks against the system by forcing validators to store large amounts of data?
- Is pallet storage culled regularly?
- How is witnessing implemented? Is it possible for the same validator to witness the same event multiple times?
- Is the custom origin defined by the `cf-witnesser` pallet implemented and enforced correctly?
- Are staked funds locked on Ethereum before they become accessible on the State Chain?
- Is it possible to claim funds staked by other users?
- Would transaction reordering on Ethereum prevent users from either staking funds or claiming staked funds?
- Could the timestamps used by the `cf-staking` and `cf-governance` pallets cause validators to disagree on transaction validity?
- Could a successful broadcast be rescheduled by mistake or replayed by a malicious validator?
- Could a broadcast be blocked if a malicious validator is nominated to submit the transaction to the external chain?
- Does the `cf-threshold-signature` pallet correctly validate threshold signatures?
- Could auction parameters change midway through an auction?
- How are swaps implemented? Could user funds become stuck or be lost during a swap?
- Is it possible to waste system resources by requesting a large number of ingress

addresses?

- How is liquidity provisioning implemented?
- Is it possible to vote multiple times for the same governance proposal?
- Is the custom origin defined by the `cf-governance` pallet implemented and enforced correctly?
- Are there arithmetic issues in the AMM implementation? How does the implementation differ from Uniswap v3?
- Does the `FLIP` smart contract implement the ERC-20 standard correctly?
- Are governance access controls implemented correctly on Ethereum?
- Could signed governance calls be replayed on Ethereum?
- Do the `Vault` and `Deposit` contracts handle nonstandard ERC-20 tokens correctly?
- Is the system vulnerable to cross-chain DoS attacks?
- Can cross-chain messaging be misused to elevate user privileges within the system?
- Does the `Vault` contract invalidate nonces correctly?
- Does the `KeyManager` contract validate threshold signatures correctly? Can signatures or messages be replayed?

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:

Witnessing. The `cf-witnesser` pallet focuses on witnessing external events on other blockchains and serves as an entry point for events on external chains. We reviewed the witnessing flow to ensure that voting is implemented correctly, that ingress witnesses cannot be replayed, and that a single validator cannot vote on the same witness multiple times. We also reviewed the implementation and use of the custom origins defined by the pallet.

Staking/claiming. When staking funds into the system, users interact with the `StakeManager` contract, specifying the target address and amount they intend to stake on the State Chain. When claiming funds, users interact with the `cf-staking` pallet, which allows them to claim their funds on Ethereum. We reviewed both these user flows from start to finish, focusing on the proper locking of funds on Ethereum before they are minted on the State Chain. We checked for impacts of transaction reordering on the system flow to ensure users cannot be blocked. We also reviewed the use of timestamps to ensure that honest validators cannot disagree on transaction validity.

Broadcasting. The `cf-broadcast` pallet is responsible for managing transactions from the State Chain to one of the supported external chains. This entails managing threshold signing ceremonies, nominating a validator to send the transaction, handling timeouts and scheduling retries, and handing over control to the calling pallet once the transaction has successfully been sent. We reviewed the entire broadcast flow to ensure that validators cannot circumvent the consensus mechanism and that different validators cannot end up with inconsistent views of the broadcast transaction. We also reviewed the use of custom origins to ensure that a call is either threshold signed or witnessed by the `cf-witnesser` pallet. Finally, we checked that timeouts are implemented correctly and that successful broadcasts are rescheduled by the pallet.

Threshold signing. The external `Vault` instances are controlled by a shared key. To sign transactions, the system uses the FROST threshold signature scheme. The signature scheme is implemented by the Chainflip engine, but the signing ceremonies are managed by the `cf-threshold-signature` pallet. We reviewed the interactions between the `cf-threshold-signature` pallet, the `cf-broadcast` pallet, and the Chainflip engine. We also reviewed the use of unsigned extrinsics and the implementation of the `ValidateUnsigned` Rust trait. We focused on how signatures are verified by the State Chain and reviewed the implementation and use of the custom origin defined by the pallet. Additionally, we checked that the timeouts are implemented correctly and that successful signatures are not rescheduled.

Epoch rotations, validator management, and auctions. The system relies on regular epoch rotations to update the authority set. This is done using an auction system. In each epoch, a new auction for the next authority set is held. We reviewed and documented the flow of an epoch rotation from the `cf-validator` pallet to the `cf-vaults` pallet. We also reviewed the different extrinsics defined by the `cf-validator` pallet. We reviewed the auctions' interactions with epoch rotations and how the validator set is updated. We checked that each auction results in a well-defined set of winners, and we considered what would happen if auction parameters were updated before an auction was over.

Ingress/egress. The `cf-ingress-egress` pallet manages the interactions between the source and target chain `Vault` instances and other State Chain pallets when funds are swapped, liquidity is provided, or a cross-chain message is sent. We manually reviewed the interactions caused by each of these operations. We focused on the interactions between the State Chain, the engine, and the external `Vault` instances to ensure they were correct. We also reviewed the use of `Deposit` contracts on Ethereum and looked for potential denial-of-service (DoS) attacks and ways to force the system to waste resources on invalid requests.

Liquidity provisioning. The `cf-lp` pallet implements the logic around adding and withdrawing liquidity to and from the system. We reviewed the extrinsics defined by the pallet, focusing on ensuring that crediting and withdrawing funds could not cause overflows. We also reviewed the user flows for adding and withdrawing liquidity as part of our review of the `cf-ingress-egress` pallet.

Governance. The `cf-governance` pallet implements functionality used by the governance council for maintaining and updating the system. We reviewed the proposal life cycle management, ensuring that the corresponding extrinsics could be called only by governance members and that members could vote only once on each proposal. We also reviewed the custom origin defined by the pallet to ensure that it was used correctly throughout the system.

Token holder governance. The `cf-tokenholder-governance` pallet implements functionality to allow token holders to propose updates of the governance and community keys on external chains. We reviewed the extrinsics defined by the pallet, as well as the flow from a new proposal to a key update. In particular, we checked that it is not possible to influence the result of a proposal by voting multiple times.

AMM. The JIT AMM implements the main swapping mechanism and supports both range orders and limit orders. The range order implementation is inspired by Uniswap v3. We reviewed all the arithmetic related to swapping funds. We reviewed the implementation for potential rounding issues and ensured that rounding is favorable to the pool and matches Uniswap v3. We reviewed the range order implementation against Uniswap v3 and noted any differences. We also reviewed the limit order implementation and the interactions between range and limit orders. Finally, we investigated potential maximum extractable value (MEV) incentives introduced by the design.

Governance (Ethereum). The `GovernanceCommunityGuarded` contract implements functionality that allows the governance council to manage aspects of the smart contracts deployed by the system. We manually reviewed the contract, focusing on the key update mechanism, and ensured that no single entity can take over the protocol and that the system can be recovered if a key is set incorrectly. We checked that access controls are implemented for all privileged functions and that signed messages cannot be replayed.

`FLIP` (Ethereum). The `FLIP` contract implements the FLIP ERC-20 token. We checked that the contract conforms to the ERC-20 standard and that privileged functions enforce proper access controls.

`StakeManager` (Ethereum). The `StakeManager` contract manages the staking of FLIP tokens. We manually reviewed the staking and claiming flows. We also reviewed the use of timestamps to ensure they cannot introduce issues.

`TokenVesting` (Ethereum). The `TokenVesting` contract implements a token vesting scheme. We manually reviewed the staking and release flows, checked that the contract enforces proper access controls, and reviewed the revocation implementation.

`Vault` (Ethereum). The `Vault` contract manages native and ERC-20 tokens held on Ethereum. It is also responsible for deploying contracts and fetching ingressed funds. We manually reviewed the access controls enforced by each function. We checked that all hashes include all relevant function parameters. Additionally, we reviewed the logic implemented for handling nonstandard ERC-20 tokens. We also reviewed nonce invalidation and the assumptions around arbitrary call execution. Finally, we investigated potential DoS attack vectors against the State Chain.

`Deposit` (Ethereum). The `Deposit` contract is used to simplify ingress for system users. We manually reviewed the ingress flow and the fetch implementation. We also looked for potential issues related to nonstandard ERC-20 tokens.

`KeyManager` (Ethereum). The `KeyManager` contract holds the aggregate and governance keys and is used to verify threshold signatures. We manually reviewed the contract, focusing on signature verification. In particular, we checked that the hash contains all relevant data. We checked for potential issues related to replay attacks and signature verification bypasses. We also checked that cases when the `ecrecover` precompile returns `address(0)` are handled correctly.

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. During this project, we were unable to perform comprehensive testing of the following system elements, which may warrant further review:

- `FLIP` emissions (which are used to distribute rewards)
- Migrations
- Docker files and build scripts
- Squid multicall used by the `Vault`

Automated Testing

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

Test Harness Configuration

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

Tool	Description	Policy
`cargo-audit`	A Cargo plugin for reviewing project dependencies for known vulnerabilities	Appendix C
`cargo-geiger`	A tool that lists statistics related to the use of unsafe Rust code in a Rust crate and all its dependencies	Appendix C
`cargo-llvm-cov`	A Cargo plugin for generating LLVM source–based code coverage	Appendix C
`cargo-outdated`	A Cargo plugin that identifies project dependencies with newer versions available	Appendix C
Clippy	An open-source Rust linter used to catch common mistakes and unidiomatic Rust code	Appendix C
Dylint	An open-source Rust linter developed by Trail of Bits to identify common code quality issues and mistakes in Rust code	Appendix C
Echidna	An Ethereum smart contract fuzzer developed by Trail of Bits	Appendix C
Semgrep	An open-source static analysis tool for finding bugs and enforcing code standards	Appendix C

Slither	A static analyzer for Solidity code developed by Trail of Bits	Appendix C
`yarn audit`	A Yarn subcommand for reviewing project dependencies for known vulnerabilities	Appendix C

Areas of Focus

Our automated testing and verification work focused on detecting the following issues:

- General code quality issues and unidiomatic code patterns
- Issues related to error handling and the use of `unwrap` and `expect`
- Extensive use of unsafe code
- Poor unit and integration test coverage
- General issues with dependency management and known vulnerable dependencies
- Smart contract invariants as outlined below

Test Results

The results of this focused testing are detailed below.

Chainflip back end. We ran several static analysis tools such as Clippy, Dylint, and Semgrep to identify potential code quality issues in the codebase. We then used the Cargo plugins `cargo-geiger`, `cargo-outdated`, and `cargo-audit` to detect the use of unsafe Rust and to review general dependency management practices. Finally, we ran `cargo-llvm-cov` to review the unit and integration test coverage for the back-end repository.

Property	Tool	Result
The project adheres to Rust best practices by fixing code quality issues reported by linters such as Clippy.	Clippy	**Passed**
The project does not contain any vulnerable code patterns identified by Dylint.	Dylint	**Passed**

The project's use of panicking functions such as `unwrap` and `expect` is limited.	Semgrep	TOB-CHFL-25
The project contains a reasonable amount of unsafe code for what the implementation is trying to achieve.	`cargo-geiger`	**Passed**
To avoid technical debt, the project continuously updates dependencies as new versions are released.	`cargo-outdated`	**Passed**
The project does not depend on any libraries with known vulnerabilities.	`cargo-audit`	TOB-CHFL-5
All components of the codebase have sufficient test coverage.	`cargo-llvm-cov`	**Passed**

Chainflip smart contracts. We ran Slither and Echidna on the Ethereum smart contracts. For Echidna, Chainflip provided a base fuzzing test suite for the Ethereum smart contracts. This suite was modified and extended to ensure that all contract functions are covered by the fuzzing tests.

The results for the provided invariants are shown in the table below.

Property	**Tool**	**Result**
`FLIP` contract reference in the `StakeManager` contract does not change.	Echidna	**Passed**
`KeyManager` references in the `StakeManager` and `Vault` contracts are kept synchronized. (This particular property requires a signed call from the aggregate key for each change. A wrapper was created to perform both changes in the same transaction to test for intended changes.)	Echidna	**Passed**

Governance key references in the `StakeManager` and `Vault` contracts are kept synchronized.	Echidna	**Passed**
Community key references in the `StakeManager` and `Vault` contracts are kept synchronized.	Echidna	**Passed**

For the FLIP token, the standard procedure of property testing with crytic/properties was followed, according to the repository documentation. The results are shown in the table below.

Property	**Tool**	**Result**
User balances must not exceed total supply.	Echidna	**Passed**
Address-zero balances must always be zero.	Echidna	**Passed**
Transfers to address zero must revert.	Echidna	**Passed**
Self-transfers must not break internal accounting.	Echidna	**Passed**
Transfers for more than the available balance must revert.	Echidna	**Passed**
Zero-amount transfers must not break internal accounting.	Echidna	**Passed**
Token transfers must update internal accounting.	Echidna	**Passed**
`Approve` must set allowances.	Echidna	**Passed**

`TransferFrom` must decrease allowances for the amount transferred.	Echidna	**Passed**
`Allowance` must be modified via `increaseAllowance` and `decreaseAllowance`.	Echidna	**Passed**
Token burning must update user balances and total supply.	Echidna	**Passed**
Token minting must update user balances and total supply.	Echidna	**Passed**

Finally, we ran `yarn audit` on the Ethereum contract repository to review the smart contract dependencies for known vulnerabilities. The result of this test is provided below.

Property	**Tool**	**Result**
The project does not depend on any libraries with known vulnerabilities.	`yarn audit`	TOB-CHFL-5

Codebase Maturity Evaluation

Trail of Bits uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

Category	Summary	Result
Arithmetic	The back end uses saturating or checked arithmetic almost exclusively when there is a risk of overflows. When ordinary (overflowing) arithmetic is used, it is typically clear from the constituent types that there is no risk of overflow. Almost all instances that require extra care with overflow analysis are documented in comments. This is particularly true for the implementation of the automated market maker (AMM). Proper analysis of precision loss due to rounding is also in place for arithmetic related to the AMM.	**Satisfactory**
Auditing	Both the Chainflip back end and the smart contracts are generally consistent about emitting events to signal that a processing step has either completed or failed. However, we did identify one issue related to the inconsistent use of events (TOB-CHFL-12). Each back-end component also implements logging to track unexpected events. This makes it easier for developers to understand how transactions flow through the system and to investigate potential incidents. There is also a system in place to monitor for failures and unexpected events on the currently deployed testnet. However, there is currently no well-defined incident response plan in place. We recommend creating an incident response plan to ensure Chainflip has a clear course of action in case vulnerabilities are found once the system is deployed.	**Moderate**
Authentication / Access Controls	The State Chain implements adequate access controls to restrict the origin of each extrinsic to a privileged subset	**Satisfactory**

	of all users. Additionally, many extrinsics can be called only via pallets that enforce a high level of consensus, such as `cf-witnesser`, `cf-governance`, or `cf-threshold-signature`. Privileged roles in the system are partially documented in the white paper. However, there are places (e.g., related to the governance and community councils) where the documentation differs from the implementation.	
Complexity Management	The State Chain codebase is well organized according to Substrate best practices. The implementation uses the Rust type system to ensure correctness. It also relies heavily on the Rust trait system and declarative macros to avoid code duplication, which reduces complexity. On the other hand, we found that much of the asynchronous code in the Chainflip engine is hard to follow. Functions are often very long, with multiple nested (sometimes asynchronous) blocks. This makes the code harder to read, test, and maintain. We also found the overall system to be very complex. Each user operation typically generates a large set of interactions between the different system components. This type of aggregate complexity could lead to security issues when the system is deployed because it makes it difficult for any one developer to understand and analyze the entire system. In particular, the deposit flow on Ethereum could benefit from being simplified.	**Moderate**
Cryptography and Key Management	The system uses the FROST threshold signature scheme to sign outgoing transactions. This provides strong security guarantees as long as more than two-thirds of the validators follow the protocol. The shared key is rotated in each epoch. Rotating the shared key regularly in this way ensures that it can easily be changed in case of a compromise. It also ensures that previous unstaked validators cannot collude to recreate the current key.	**Strong**

Decentralization	Privileged actors in the system display a high level of decentralization. Witnessing requires a supermajority vote to invoke State Chain functionality. To enact changes on external chains, a threshold signature is required, which similarly needs an honest supermajority from the authority set to complete. Finally, the State Chain governance function requires a majority vote to enact new proposals. On the other hand, many system parameters related to decentralization (such as the size of the authority set and governance itself), as well as the runtime, can be updated by the governance council, which is controlled by Chainflip.	**Moderate**
Documentation	The Chainflip white paper provides a good high-level system overview. The white paper also contains a section on security considerations and potential risks to the system. Additionally, Chainflip provided access to their internal documentation, which was useful in understanding the different system components and various implementation design choices. We found the codebase to be generally well documented. However, there are a few instances (like the `cf-broadcast` pallet) where we found that the existing documentation differs from the actual implementation. We did not find any public documentation of the JIT AMM, the interactions between range orders and limit orders, and the differences between range orders and Uniswap v3. This means that it is not possible to verify the implementation against a given specification, which would provide stronger guarantees of correctness.	**Moderate**
Front-Running Resistance	Although some functions in the Ethereum contracts are susceptible to front-running and can be executed by any user, most instances do not appear to present vulnerabilities. However, we did identify one high-severity issue related to cross-chain communication for staking (TOB-CHFL-23). Overall, we believe the protocol would benefit from giving more consideration to front-running. Additionally, potential changes in incentives pertaining to the maximum extractable value (MEV), which may arise	**Further investigation required**

	due to the differences between the AMM and Uniswap v3, should be further investigated.	
Low-Level Manipulation	Low-level manipulations are largely avoided. However, we discovered one issue related to decoding the return data from a low-level call before verifying its length (TOB-CHFL-8). This should have been handled by a trusted library.	**Moderate**
Maintenance	The project is well maintained, and the team takes a considered approach to dependency management. For example, they maintain their own fork of Substrate to selectively merge bug fixes as they become available. However, when we reviewed the back-end and smart contract dependencies, we found several dependencies with known vulnerabilities (TOB-CHFL-5). The main README file in the back-end repository contains information about how to audit dependencies with known issues, but tools such as `cargo-audit` and `yarn audit` are not run by the CI pipeline.	**Moderate**
Memory Safety and Error Handling	The back end contains a very small amount of unsafe code. Errors are generally logged and propagated to the caller. We found a few instances where errors were silently dropped without any justification in the implementation of the Chainflip engine. However, none of these constituted a vulnerability in the system. In cases where `unwrap` and `expect` are used, it is typically clear why the call should not fail. However, we identified one issue where a user-controlled value could cause the engine to panic because of a call to `unwrap` (TOB-CHFL-25).	**Satisfactory**
Testing and Verification	The project has an extensive unit and integration test suite that tests both the happy path and various failure cases and adversarial inputs. Tests are run as part of Chainflip's CI pipeline. Test coverage data is also generated using `cargo-llvm-cov` during CI. The overall test coverage is acceptable but could be improved, and	**Moderate**

	there are some functions that are completely untested. The codebase would also benefit from introducing automated testing such as fuzz testing or property testing for critical components like the AMM. Extending the test suite with automated testing would exercise the implementation on unexpected and invalid inputs, ensuring that there are no edge cases that are missed by the current test suite. Differential fuzzing tests on the Substrate-based JIT AMM and Uniswap would also be highly beneficial to increase confidence in the implementation.	

Summary of Findings

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

ID	Title	Type	Severity
1	Step 2 of the handover protocol can be forged	Cryptography	**Medium**
2	Hash function is used as KDF in handover protocol	Cryptography	**Informational**
3	Ceremony participants can send many delayed messages	Data Validation	**Informational**
4	Binding value can be zero	Data Validation	**Informational**
5	The Chainflip back end and smart contracts have vulnerable dependencies	Patching	**Medium**
6	Potential panic in KeyId::from_bytes	Data Validation	**Informational**
7	Solidity compiler optimizations can be problematic	Undefined Behavior	**Undetermined**
8	ERC-20 token transfer fails for certain tokens	Data Validation	**High**
9	addGasNative is missing check for nonzero value	Data Validation	**Informational**
10	StakeManager contains unnecessary receive function	Data Validation	**Informational**
11	Missing events for important operations	Data Validation	**Low**
12	Nonstandard ERC-20 tokens get stuck when depositing	Data Validation	**High**

13	transfer can fail due to a fixed gas stipend	Data Validation	**Informational**
14	Low number of block confirmations configured for external blockchains	Configuration	**Undetermined**
15	Hard to diagnose error from default behavior during signer nomination	Data Validation	**Informational**
16	Failed broadcast nominees are not punished if epoch ends during broadcast	Error Reporting	**Low**
17	Nominated broadcast signer does not always report failures in engine	Error Reporting	**Informational**
18	Threshold signature liveness protection does not account for previously punished validators	Data Validation	**Informational**
19	A malicious minority can ruin liveness	Denial of Service	**Medium**
20	Validators can report nonparticipants in ceremonies	Data Validation	**Medium**
21	Staker funds can be locked via front-running	Denial of Service	**High**
22	Unbounded loop execution may result in out-of-gas errors	Configuration	**Informational**
23	Anyone can cause the Chainflip engine to panic	Denial of Service	**Medium**
24	Failed deposits are incorrectly witnessed as having succeeded	Data Validation	**High**
25	Validators are not reimbursed for transactions submitted to external chains	Data Validation	**Low**
26	MEV incentives are unclear and require further investigation	Undefined Behavior	**Undetermined**

Detailed Findings

1. Step 2 of the handover protocol can be forged	
Severity: **Medium**	Difficulty: **Low**
Type: Cryptography	Finding ID: TOB-CHFL-1
Target: Bitcoin key handover protocol	

Description

In step 2 of the handover protocol, the participant $\mathbb{B}$ needs to signal agreement that participant $\mathbb{A}$ performed step 1 correctly and that $\mathbb{B}$ has correctly recovered the ephemeral key t. Step 2 currently comprises a Schnorr-like proof of knowledge of $\mathbb{B}$'s secret share b, made noninteractive using the Fiat-Shamir transform:

$$N = g^n$$
$$z = n - bH(l||T)$$

However, the hash calculation does not include the commitment N, which means that anyone can generate a random z and compute $N = g^z B^{H(l||T)}$. Therefore, this does not actually prove knowledge of b.

Exploit Scenario

Participant $\mathbb{A}$ does not execute step 1 correctly (such that participant $\mathbb{B}$ cannot obtain t), generates a random z, and computes $N = g^z B^{H(l||T)}$. Before $\mathbb{B}$ has a chance to complain about step 1, $\mathbb{A}$ jumps to step 3 and publishes $k = t - a$. When confronted with the complaint about step 1, $\mathbb{A}$ publishes z and N and claims that $\mathbb{B}$ has signaled agreement in step 2. Since k has already been published, $\mathbb{A}$ claims that, to allow the rest of the validators to verify step 1, they cannot publish t because it would reveal secret share a to all validators. It is not clear to the validators whether $\mathbb{A}$ or $\mathbb{B}$ should be slashed.

Because this may result in the slashing of $\mathbb{B}$, we have set the severity to medium. Since the flaw in the proof of knowledge is relatively well known, we have set the difficulty of exploitation to low. However, there may be mitigating factors—not directly part of the protocol—that increase the difficulty of exploitation, such as the infrastructure underlying the broadcast protocol and peer-to-peer communication.

Recommendations

Short term, include the commitment N in the hash calculation (*i.e.*, $z = n - bH(N||l||T)$ to be consistent with a regular Schnorr signature of the message $l||T$.

Long term, use the alternative handover proposal that comprises a resharing of the current key to new participants.

2. Hash function is used as KDF in handover protocol	
Severity: **Informational**	Difficulty: **Not Applicable**
Type: Cryptography	Finding ID: TOB-CHFL-2
Target: Bitcoin key handover protocol	

Description

In step 1 of the key handover protocol, participant $\mathbb{A}$ performs a DH key exchange with participant $\mathbb{B}$'s public key and applies a cryptographic hash function to the resulting shared secret. The resulting value is used as a keystream to encrypt the ephemeral key t as if using a stream cipher. Therefore, the hash function is used as a key derivation function (KDF). However, cryptographic hash functions are not necessarily designed to be used as KDFs, so it is recommended to use a dedicated KDF, such as HKDF, instead.

Recommendations

Short term, replace the hash function in step 1 with HKDF, which can be instantiated using the same hash function that is currently proposed for step 1.

Long term, consider using encryption with an established symmetric scheme (such as an AEAD like `XSalsa20+Poly1305`), rather than a plain XOR with key material.

3. Ceremony participants can send many delayed messages

Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-CHFL-3
Target: `engine/src/multisig/client/ceremony_runner.rs`	

Description

The ceremony runner allows participants to send ceremony data that arrives at the runner one round early. Each message received by the runner is processed via `CeremonyRunner::process_or_delay_message`, and if the message data is found to correspond to the round after the current round, the delayed messages map is updated with the new ceremony data in `CeremonyRunner::add_delayed`.

While allowing messages to arrive one round early is a unique idea, this would permit malicious participants to send many messages containing ceremony data for the next round, which would cause the runner to discard the previously stored data. This is not a vulnerability per se, but generally, if a participant sends multiple messages for future rounds of the protocol, it is a strong indication of malicious behavior, and we see no reason to allow it.

```
fn add_delayed(&mut self, id: AccountId, m: Ceremony::Data) {
        // ...
        self.delayed_messages.insert(id, m);
}
```

Figure 3.1: The `add_delayed` function calls `BTreeMap::insert`, which updates entries.
(`engine/src/multisig/client/ceremony_runner.rs`)

Recommendations

Short term, consider logging when the `insert` operation updates a key, rather than performing a new insertion.

Long term, consider being stricter about delayed messages and disallowing participants to send updated delayed messages.

4. Binding value can be zero

Severity: **Informational**	Difficulty: **Not Applicable**
Type: Data Validation	Finding ID: TOB-CHFL-4
Target: `engine/src/multisig/client/signing/signing_detail.rs`	

Description

To protect against the attack described by Drijvers et al. in the paper On the Security of Two-Round Multi-Signatures, FROST requires each participant to derive a binding value $\rho_i = H_1(i, m, B)$, which binds the message, the set of participants, and the set of commitments to the signature share. To ensure that ρ_i is nonzero, the paper requires the hash function H_1 to take values in the multiplicative group $\mathbb{Z}_q^*$.

In the implementation, H_1 is computed by the `gen_rho_i` function, which simply hashes the index of the current signer, the message, and the commitments in B and then reduces the resulting digest modulo the curve order. Since the function does not check that the result is in the multiplicative group $\mathbb{Z}_q^*$, the output may be zero. In practice, this happens with vanishingly small probability.

```
fn gen_rho_i<P: ECPoint>(
	index: AuthorityCount,
	msg: &[u8],
	signing_commitments: &BTreeMap<AuthorityCount, SigningCommitment<P>>,
	all_idxs: &BTreeSet<AuthorityCount>,
) -> P::Scalar {
	let mut hasher = Sha256::new();
	hasher.update(b"I");
	hasher.update(index.to_be_bytes());
	hasher.update(msg);

	// This needs to be processed in order!

	for idx in all_idxs {
		let com = &signing_commitments[idx];
		hasher.update(idx.to_be_bytes());
		hasher.update(com.d.as_bytes());
		hasher.update(com.e.as_bytes());
	}

	let result = hasher.finalize();
```

```
        let x: [u8; 32] = result.as_slice().try_into().expect("Invalid hash size");

        P::Scalar::from_bytes_mod_order(&x)
}
```

Figure 4.1: The `gen_rho_i` function does not check that the return value is nonzero. (`engine/src/multisig/client/signing/signing_detail.rs:84-109`)

Recommendations

Short term, ensure that the return value from `gen_rho_i` lies in the range $[1, q)$.

Long term, review the implementation against the FROST paper to ensure that all values are sampled from the correct sets.

5. The Chainflip back end and smart contracts have vulnerable dependencies	
Severity: **Medium**	Difficulty: **High**
Type: Patching	Finding ID: TOB-CHFL-5
Target: Chainflip back end and smart contracts	

Description

Both the Chainflip back end and the Ethereum smart contract repositories contain vulnerabilities that can result in information disclosure and other side effects.

The Chainflip back-end codebase uses the following vulnerable Rust dependencies. The `openssl` crate is a dependency of `web3`, and `remove_dir_all` and `time` crates are both indirect dependencies of Substrate.

Dependency	Version	ID	Description
`openssl`	0.10.45	RUSTSEC-2023-0022	`X509NameBuilder::build` returned object is not thread safe
`openssl`	0.10.45	RUSTSEC-2023-0023	`SubjectAlternativeName` and `ExtendedKeyUsage::other` allow arbitrary file read
`openssl`	0.10.45	RUSTSEC-2023-0024	`X509Extension::new` and `X509Extension::new_nid` null pointer dereference
`remove_dir_all`	0.5.3	RUSTSEC-2023-0018	Race condition enabling link following and time-of-check time-of-use
`time`	0.1.45	RUSTSEC-2020-0071	Potential segfault in the `time` crate
`ansi_term`	0.12.1	RUSTSEC-2021-0139	Unmaintained
`mach`	0.3.2	RUSTSEC-2020-0168	Unmaintained
`parity-util-mem`	012.0	RUSTSEC-2022-0080	Unmaintained
`parity-wasm`	0.45.0	RUSTSEC-2022-0061	Unmaintained

Table 5.1: Back-end dependencies with known vulnerabilities

The identified vulnerabilities in `openssl` are all related to X.509 certificate parsing in TLS. The crate is included as a dependency of the `web3` and `reqwest` crates.

- `web3`: The `web3` transport implementation is used for RPC over either HTTP or WebSockets. There is nothing in the implementation preventing a user from using either HTTPS or secure WebSockets, which would expose the attack surface in `openssl`. Since the RPC interfaces are typically exposed only locally on the host, we consider the risk associated with such exposure to be less severe.

- `reqwest`: The `reqwest` crate is used only during unit testing of the `HealthChecker` type defined by the Chainflip engine.

According to the web3 documentation, it is possible to use Rustls as a drop-in replacement for `openssl` by enabling the feature `http-rustls-tls`.

The vulnerability in `remove_dir_all` requires the attacker to have write access to the local filesystem.

The `time` crate is a dependency of `chrono`. However, according to CVE-2020-26235 in the `chrono` repository, the warning is a false positive because `chrono` does not rely on the vulnerable APIs in `time`.

By running `yarn audit` on the Ethereum smart contract repository, we identified 34 vulnerable dependencies. These include the following packages with known CVEs:

Dependency	Version	ID	Description
`@openzeppelin/contracts`	< 4.4.1	CVE-2021-46320	Improper initialization in OpenZeppelin
`minimist`	< 1.2.6	CVE-2021-44906	Prototype pollution in `minimist`

Table 5.2: Smart contract dependencies with known vulnerabilities

Exploit Scenario

A node operator chooses to rely on an external RPC endpoint like Infura to monitor blockchain events. The API is served over HTTPS, so the attack surface in `openssl` is exposed on an external interface on the host. An attacker with a privileged network position notices this and manages to exploit RUSTSEC-2023-0023 to read sensitive files, such as the State Chain private key file, over the network.

Recommendations

Short term, take the following actions:

- Replace `openssl` with Rustls by enabling the `web3` feature `http-rustls-tls`.

- Update vulnerable dependencies to ensure code is not affected by vulnerable package dependencies.

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

6. Potential panic in KeyId::from_bytes

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-CHFL-6
Target: `state-chain/primitives/src/lib.rs`	

Description

The `KeyId::from_bytes` method is used to deserialize key IDs when keys are read from persistent storage. The method will panic if the input is shorter than the value of the `size_of_epoch_index` variable.

```
pub fn from_bytes(bytes: &[u8]) -> Self {
        let size_of_epoch_index = sp_std::mem::size_of::<EpochIndex>();
        let epoch_index = EpochIndex::from_be_bytes(
            bytes[..size_of_epoch_index].try_into().unwrap()
        );
        let public_key_bytes = bytes[size_of_epoch_index..].to_vec();
        Self { epoch_index, public_key_bytes }
}
```

Figure 6.1: The `KeyId::from_bytes` method panics on short inputs.
(`state-chain/primitives/src/lib.rs:133-139`)

Exploit Scenario

A Chainflip developer reuses `KeyId::from_bytes` to deserialize key IDs from an untrusted source. This opens the node to denial-of-service attacks from malicious users.

Recommendations

Short term, consider replacing `KeyId::from_bytes` with an implementation of `TryFrom<&[u8]>`, which returns an error if the conversion fails.

Long term, document panicking behavior throughout the codebase. Use fuzzing to identify edge cases that may cause the implementation to panic.

7. Solidity compiler optimizations can be problematic

Severity: **Undetermined**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-CHFL-7
Target: Solidity codebase	

Description

Chainflip has enabled optional compiler optimizations in Solidity. According to a November 2018 audit of the Solidity compiler, the optional optimizations may not be safe.

Solidity compiler optimizations are disabled by default, and it is unclear how many contracts in the wild use them. Therefore, it is unclear how well they are being tested and exercised. Moreover, optimizations are actively being developed.

High-severity security issues due to optimization bugs have occurred in the past. A high-severity bug in the `emscripten`-generated `solc-js` compiler used by Truffle and Remix persisted until late 2018; the fix for this bug was not reported in the Solidity changelog. Another high-severity optimization bug resulting in incorrect bit shift results was patched in Solidity 0.5.6. More recently, another bug due to the incorrect caching of Keccak-256 was reported.

It is likely that there are latent bugs related to optimization and that future optimizations will introduce new bugs.

Exploit Scenario

A latent or future bug in Solidity compiler optimizations or in the Emscripten transpilation to `solc-js` causes a security vulnerability in the Chainflip contracts.

Recommendations

Short term, measure the gas savings from optimizations and carefully weigh them against the possibility of optimization-related bugs.

Long term, monitor the development and adoption of Solidity compiler optimizations to assess their maturity.

8. ERC-20 token transfer fails for certain tokens

Severity: **High**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-CHFL-8
Target: `contracts/Vault.sol`	

Description

Due to an incorrect check, nonstandard ERC-20 tokens that do not return a Boolean value when transferred could be stuck in the protocol. USDT is an example of such a token.

When tokens are transferred from the `Vault`, the internal `_transfer` function is called. This function aims to cover nonstandard ERC-20 tokens that do not return a Boolean value on a successful transfer and works like OpenZeppelin's `safeTransfer` function. The function checks the call success status and requires that either no data is returned or that the data must be decodable to the Boolean value `true`.

```
// solhint-disable-next-line avoid-low-level-calls
(bool success, bytes memory returndata) = token.call(
    abi.encodeWithSelector(IERC20(token).transfer.selector, recipient, amount)
);

// No need to check token.code.length since it comes from a gated call
bool transferred = success && (
    abi.decode(returndata, (bool)) || returndata.length == uint256(0)
);
if (!transferred) emit TransferTokenFailed(recipient, amount, token, returndata);
```

Figure 8.1: The `_transfer` function (`contracts/Vault.sol:243`)

Unlike the implementation by OpenZeppelin, however, the `_transfer` function tries to decode the return data first, without checking its length beforehand. Because this check is performed first, the function will revert for tokens that do not return any data.

All functions that transfer funds from the user to the `Vault` use OpenZeppelin's `safeTransferFrom` function and are not affected by this issue. This issue is present only when transferring funds out of the `Vault` via `transfer`, `transferBatch`, and `govWithdraw`.

However, because there are ways to recover tokens that would otherwise be stuck, the impact is minimized. The `executexSwapAndCall` and `executeActions` functions use OpenZeppelin's `safeTransfer` function and could be used to transfer funds from the `Vault` via governance.

Currently, Chainflip supports only a limited number of tokens, which are not affected by this issue. However, it could become a problem once more tokens are supported.

Exploit Scenario

Chainflip adds support for the USDT token. Alice initiates a swap and deposits 1M USDT. Due to the incorrect check, the tokens end up stuck in the protocol.

Recommendations

Short term, change the order of the Boolean expressions so that a return data length of zero will be checked first, or simply use OpenZeppelin's `safeTransfer` function.

Long term, make sure that all edge cases have sufficient test coverage, especially when designing a protocol that is meant to handle nonstandard ERC-20 tokens.

9. addGasNative is missing check for nonzero value

Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-CHFL-9
Target: `contracts/Vault.sol`	

Description

The `addGasNative` function, which adds additional gas to a call, does not check for nonzero `msg.value` amounts.

```
function addGasNative(bytes32 swapID) external payable override onlyNotSuspended {
    emit AddGasNative(swapID, msg.value);
}
```

Figure 9.1: The `addGasNative` function (`contracts/Vault.sol`)

This contrasts with all other functions that contain checks for nonzero values using the `nzUint` modifier. This issue occurs in the `addGasToken` function, for example.

```
function addGasToken(
    bytes32 swapID,
    uint256 amount,
    IERC20 token
) external override onlyNotSuspended nzUint(amount) {
    token.safeTransferFrom(msg.sender, address(this), amount);
    emit AddGasToken(swapID, amount, address(token));
}
```

Figure 9.2: The `addGasToken` function (`contracts/Vault.sol`)

Recommendations

Short term, include the modifier `nzUint(msg.value)` to check for nonzero amounts.

Long term, make sure all functions are sufficiently tested and adhere to the same assumptions and requirements where applicable.

10. StakeManager contains unnecessary receive function	
Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-CHFL-10
Target: `contracts/StakeManager.sol`	

Description

The `StakeManager` contract contains a `receive` function, despite not requiring ether interactions.

Contracts without `receive` and/or `fallback` functions will automatically reject unexpected ether sent with a transaction. Because the `receive` function is present in this codebase, users may accidentally send ETH to the `StakeManager` contract, which, instead of reverting the transaction by default, will accept the ETH:

```
/**
 *  @notice Allows this contract to receive native tokens
 */
receive() external payable {}
```

Figure 10.1: The `receive` function (`contracts/StakeManager.sol`)

The only way to retrieve the ETH is for the governor to call the `govWithdrawNative` function.

Recommendations

Short term, remove the `receive` function from the `StakeManager` contract.

Long term, always keep functions in the contracts to a minimum, and ensure that unused functions are removed.

11. Missing events for important operations

Severity: **Low**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-CHFL-12
Target: `contracts/Vault.sol` and `contracts/KeyManager.sol`	

Description

Unlike most other functions in the `Vault` contract, the `executexCall` and `executexSwapAndCall` functions do not emit events.

```
function executexCall(
    SigData calldata sigData,
    address recipient,
    uint32 srcChain,
    bytes calldata srcAddress,
    bytes calldata message
)
    external
    override
    onlyNotSuspended
    nzAddr(recipient)
    consumesKeyNonce(
        // ...
    )
{
    ICFReceiver(recipient).cfReceivexCall(srcChain, srcAddress, message);
}
```

Figure 11.1: The `executexCall` function (`contracts/Vault.sol`)

It is unclear whether events are required to track the execution of this call.

Furthermore, the `govWithdrawNative` function in the `KeyManager` contract is missing the `GovernanceWithdrawal` event, which is included in the same function in the `StakeManager` contract.

```
function govWithdrawNative() external override onlyGovernor {
    uint256 amount = address(this).balance;

    // Could use msg.sender but hardcoding the get call just for extra safety
    address recipient = _getGovernanceKey();
    payable(recipient).transfer(amount);
}
```

Figure 11.2: The `govWithdrawNative` function (`contracts/KeyManager.sol`)

Recommendations

Short term, include events that are emitted in the case of a successful (or unsuccessful) call.

Long term, ensure all functions properly emit events. Use Slither to help identify all instances.

12. Nonstandard ERC-20 tokens get stuck when depositing

Severity: **High**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-CHFL-13
Target: `contracts/Deposit.sol`	

Description

Nonstandard ERC-20 tokens, such as USDT, are not retrievable by the `Deposit` contract because it does not handle the `transfer` function correctly.

The `Deposit` contract is meant to fetch tokens sent by a depositor. Tokens are fetched when the contract is constructed or when the `fetch` function is called.

```
function fetch(IERC20Lite token) external {
    require(msg.sender == vault);

    // Slightly cheaper to use msg.sender instead of Vault.
    if (address(token) == 0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE) {
        // solhint-disable-next-line avoid-low-level-calls
        (bool success, ) = msg.sender.call{value: address(this).balance}("");
        require(success);
    } else {
        // Not checking the return value to avoid reverts for tokens with no return
value.
        token.transfer(msg.sender, token.balanceOf(address(this)));
    }
}
```

Figure 12.1: The `fetch` function (`contracts/Deposit.sol`)

The comment in figure 13.1 suggests that not checking the return value will suffice to cover nonstandard tokens that do not return a Boolean value when calling `transfer`. This is not true, however. The contract uses the `IERC20Lite` interface when calling `token.transfer`.

```
interface IERC20Lite {
    // Taken from OZ:
    /**
     * @dev Moves `amount` tokens from the caller's account to `recipient`.
     *
     * Returns a boolean value indicating whether the operation succeeded.
     *
     * Emits a {Transfer} event.
     */
    function transfer(address, uint256) external returns (bool);

    // ...
}
```

Figure 12.2: The `IERC20Lite` interface (`contracts/interfaces/IERC20Lite.sol`)

Since the interface specifies that a Boolean return value is expected, the Solidity compiler automatically includes a check on the function's return data size and reverts if it is not at least 32 bytes. This means that the tokens will end up stuck in the contract.

Exploit Scenario

Chainflip adds support for the USDT token. Alice deposits 1M USDT tokens to a designated address. The Chainflip protocol is unable to fetch the deposited tokens due to the bug in the `transfer` call. The tokens are now stuck forever.

Recommendations

Short term, use OpenZeppelin's `safeTransfer` function, or change the function signature in `IERC20Lite` to `function transfer(address, uint256) external` so that a return value is not expected. This will ensure that Solidity omits any check on the return value.

Long term, include sufficient test coverage for nonstandard tokens when aiming to integrate them.

13. transfer can fail due to a fixed gas stipend

Severity: **Informational**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-CHFL-14
Target: `contracts/KeyManager.sol`	

Description

The `govWithdrawNative` function transfers ether from a `Vault` instance to the governor via the `recipient.transfer` method. The `transfer` function executes the call with a fixed gas stipend of 2300. This gas stipend allows only limited execution when receiving ether. If the recipient is a proxy contract, for example, this call will fail. Because the recipient is trusted, there is no reason to allow this function to fail due to out-of-gas errors.

```
function govWithdrawNative() external override onlyGovernor {
    uint256 amount = address(this).balance;

    // Could use msg.sender or getGovernor() but hardcoding the get call just for
extra safety
    address recipient = getKeyManager().getGovernanceKey();
    payable(recipient).transfer(amount);
    emit GovernanceWithdrawal(recipient, amount);
}
```

Figure 13.1: The `govWithdrawNative` function (`contracts/KeyManager.sol`)

Recommendations

Short term, have the code use `recipient.call{value: amount}(“”)` to transfer ether while forwarding all remaining gas to the recipient. In general, using `.call` requires extra care because it could allow reentrancy attacks. In this case, `.call` can be used because it is a privileged call that follows the Checks-Effects-Interactions pattern.

Long term, avoid using the `transfer` function to transfer ether.

14. Low number of block confirmations configured for external blockchains

Severity: **Undetermined**	Difficulty: **High**
Type: Configuration	Finding ID: TOB-CHFL-15
Target: `state-chain/runtime/src/constants.rs`	

Description

The Chainflip engine requires four block confirmations for Ethereum and three block confirmations for Bitcoin before accepting their associated transaction hashes as valid.

This poses a risk when bridging from Ethereum to Substrate. The State Chain assumes the transaction has been fully settled on the external blockchain after these confirmation blocks are created. There is a real risk of accepting uncle blocks as finalized, which means that transactions on Ethereum can be witnessed and acted upon on the State Chain, despite the corresponding Ethereum blocks being reverted afterwards.

These specific numbers of required block confirmations are configured in the State Chain constants:

```
pub mod eth {
	use cf_chains::{eth::Ethereum, Chain};

	/// Number of blocks to wait until we deem the block to be safe.
	pub const BLOCK_SAFETY_MARGIN: <Ethereum as Chain>::ChainBlockNumber = 4;
}

pub mod btc {
	use cf_chains::{btc::Bitcoin, Chain};

	/// Number of blocks to wait for until we deem a BTC ingress to be safe.
	pub const INGRESS_BLOCK_SAFETY_MARGIN: <Bitcoin as Chain>::ChainBlockNumber =
3;
}
```

Figure 14.1: The block confirmation safety margins defined for Ethereum and Bitcoin (`state-chain/runtime/src/constants.rs`)

The chosen number of block confirmations should be the result of risk analysis per blockchain. These analyses should consider consensus mechanisms to create transactions, average block times, and the way each blockchain handles orphaned, uncle, and stale blocks.

Exploit Scenario

Eve submits a transaction to stake FLIP tokens through the `StakeManager` contract in Ethereum. Her transaction is picked up by a validator whose blocks have diverged and who is building off a separate fork of the Ethereum chain. Four blocks after the transaction has been settled, the witnessing process starts. Once witnessed, the `stake` function is invoked, which adds staked funds to Eve's specified account.

Afterward, the Ethereum block reverts and all transactions inside the block are dropped, reverting the transaction to stake FLIP tokens. As a result, Eve now has a stake value recorded on the State Chain but no stake locked on Ethereum.

Recommendations

Short term, process only blocks that are considered finalized by the Ethereum proof-of-stake (PoS) consensus mechanism. Finalized blocks can be requested via the RPC interface. Waiting a specific number of blocks/slots does not suffice for PoS.

Long term, execute in-depth analysis to determine safe confirmation bounds for all blockchain integrations.

References

- Proof-of-stake (PoS), Ethereum Development Documentation, last updated May 12, 2023—block finalization

- Stacy Elliott, Ethereum Beacon Chain Suffers Longest Blockchain "Reorg" in Years, May 25, 2022—seven-block reorganization

- Vitalik Buterin, On Slow and Fast Block Times, September 14, 2015—analysis on block times

- Vitalik Buterin, Ethereum Whitepaper, 2014 (last updated June 15, 2023)—analysis on uncle blocks and their ancestors

15. Hard to diagnose error from default behavior during signer nomination

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-CHFL-16
Target: `state-chain/runtime/src/chainflip/signer_nomination.rs`	

Description

When a new threshold signature is requested, the `cf-threshold-signature` pallet generates a set of signers using the `RandomSignerNomination::threshold_nomination_with_seed` method, which returns a random subset of the authority set.

```
fn threshold_nomination_with_seed<H: Hashable>(
    seed: H,
    epoch_index: EpochIndex,
) -> Option<BTreeSet<Self::SignerId>> {
    try_select_random_subset(
        seed_from_hashable(seed),
        cf_utilities::success_threshold_from_share_count(
            Validator::authority_count_at_epoch(epoch_index).unwrap_or_default(),
        ) as usize,
        eligible_authorities(
            epoch_index,
            &Reputation::validators_suspended_for(&[
                Offence::ParticipateSigningFailed,
                Offence::MissedAuthorshipSlot,
                Offence::MissedHeartbeat,
            ]),
        ),
    )
}
```

Figure 15.1: The authority count is a u32, which defaults to `0`.
(`state-chain/runtime/src/chainflip/signer_nomination.rs`)

However, the value of `Validator::authority_count_at_epoch(epoch_index)` is `None` if the `EpochAuthorityCount` storage is not set. This means that the `n` parameter for the `try_select_random_subset` function defaults to one, (since the value of `success_threshold_from_share_count(0)` is one). Thus, a single random signer is selected (even though the threshold for the key may be greater than one). This is not detected by the pallet, and a signing request with a singleton `signatories` set is sent to the engine.

In the engine, the chosen signer's `CeremonyManager` invokes the `on_request_to_sign` method, which generates a standard Schnorr signature over the payload using the signer's key share by invoking `CeremonyManager::single_party_signing`.

```
if signers.len() == 1 {
    let _result =
        result_sender.send(Ok(self.single_party_signing(payloads, key_info, rng)));
    return
}
```

Figure 16.2: The engine generates a standard Schnorr signature with its share of the secret key if it is the sole signer. (`engine/src/multisig/client/ceremony_manager.rs`)

The signer then creates a new unsigned `signature_success` extrinsic that is finally rejected by the `cf-threshold-signature` pallet implementation of the `validate_unsigned` method once the signature is validated against the shared public key.

Exploit Scenario

The `cf-validator` pallet is refactored, and an edge case, which allows the `EpochAuthorityCount` storage to be uninitialized under some circumstances, is introduced. This causes the engine to generate invalid threshold signatures in rare situations, which is difficult to reproduce and diagnose.

Recommendations

Short term, have the `threshold_nomination_from_seed` function return `None` if the `authority_count_at_epoch` function returns `None`. This will cause the `cf-threshold-signature` pallet to abort with a `SignersUnavailable` event if the authority count is unavailable.

Long term, use the `#[cfg(test)]` annotation to ensure that code paths that are used only for testing are not included in release builds.

16. Failed broadcast nominees are not punished if epoch ends during broadcast

Severity: **Low**	Difficulty: **High**
Type: Error Reporting	Finding ID: TOB-CHFL-17
Target: `state-chain/pallets/cf-broadcast/src/lib.rs`	

Description

The broadcast mechanism is used to broadcast transactions to external chains. The first phase of the broadcast protocol is to obtain consensus from a supermajority of validators from the authority set by creating a threshold signature over the transaction to be broadcast. Validators that fail to correctly execute any stage of the broadcast protocol are punished.

In phase two of the broadcast protocol, a pseudo-random validator from the authority set is nominated to sign the threshold signature and submit it to the external chain as a transaction. If the submission fails, the nominated validator is added to the list of `FailedBroadcasters` in storage and a new attempt is made with a new nominee. When the signature is eventually successfully sent (resulting in a `SignatureAccepted` event that can be witnessed by validators), all `FailedBroadcasters` are reported.

```
if let Some(failed_signers) = FailedBroadcasters::<T, I>::get(broadcast_id) {
	T::OffenceReporter::report_many(
		PalletOffence::FailedToBroadcastTransaction,
		&failed_signers,
	);
}
```

Figure 16.1: `FailedBroadcasters` are punished when the signature is accepted. (`state-chain/pallets/cf-broadcast/src/lib.rs`)

The authority set is replaced at the start of each new epoch. If the next epoch starts before the signature is successfully sent, the aggregate key is updated, so the threshold signature is no longer valid. Upon the inevitable failure, the `cf-broadcast` pallet restarts the broadcast protocol to generate a new threshold signature.

```
// If the signature verification fails, we want
// to retry from the threshold signing stage.
else {
	let callback = RequestCallbacks::<T, I>::get(broadcast_id);
	Self::clean_up_broadcast_storage(broadcast_id);
```

```
        Self::threshold_sign_and_broadcast(api_call, callback);
```

Figure 16.2: The broadcast procedure is restarted from scratch. (`state-chain/pallets/cf-broadcast/src/lib.rs`)

In this case, the `clean_up_broadcast_storage` function empties the list of `FailedBroadcasters`, who are not punished as a result. Therefore, nominated validators who want to delay certain broadcast transactions near the end of an epoch can do so without punishment.

Exploit Scenario

A malicious validator is nominated to sign and submit a broadcast transaction to the external chain near the end of an epoch. The validator chooses to delay the transaction and waits for the timeout. Since timeout occurs after the epoch is updated, the threshold signature is no longer valid. Thus, the broadcast process is restarted without punishing the validator.

Because the impact of this attack is limited to delaying the transfer of user funds, the severity is set to low. Because the validator needs to be pseudo-randomly selected at the right time of the epoch for a particular transaction they want to delay, the difficulty is set to high.

Recommendations

Short term, add punishment for `FailedBroadcasters` to the cases where threshold signature verification fails during broadcast retries.

Long term, consider documenting the different outcomes of the various ceremonies to ensure that all failure cases are handled uniformly.

17. Nominated broadcast signer does not always report failures in engine

Severity: **Informational**	Difficulty: **Not Applicable**
Type: Error Reporting	Finding ID: TOB-CHFL-18
Target: `engine/src/state_chain_observer/sc_observer/mod.rs`	

Description

The Chainflip engine does not report transaction broadcast failures to the State Chain, which forces validators to wait for a timeout before attempting to rebroadcast the corresponding transaction.

The State Chain observer in the Chainflip engine watches the State Chain for new events. When it sees a `TransactionBroadcastRequest` from the State Chain Ethereum broadcaster, it checks who the nominated signer is. If it is the current node, the engine attempts to sign and transmit an Ethereum transaction.

If signing fails, it calls the `transaction_signing_failure` extrinsic of the `cf-broadcast` pallet, informing the State Chain that a new signer should be nominated.

```
Err(e) => {
    error!("TransactionSigningRequest {broadcast_attempt_id} failed: {e:?}");

    let _result = state_chain_client.submit_signed_extrinsic(
        state_chain_runtime::RuntimeCall::EthereumBroadcaster(
            pallet_cf_broadcast::Call::transaction_signing_failure {
                broadcast_attempt_id,
            },
        ),
    ).await;
}
```

Figure 17.1: If signing fails, it is reported back to the State Chain.
(`engine/src/state_chain_observer/sc_observer/mod.rs`)

If the signing succeeds, the engine attempts to transmit the signed transaction to the Ethereum network. If this transmission fails, however, it does not inform the State Chain, which forces other validators to wait for a timeout.

```
Err(e) => {
    info!("TransmissionRequest {broadcast_attempt_id} failed: {e:?}");
},
```

Figure 17.2: If the Ethereum RPC API call fails, it is not reported back to the State Chain. (`engine/src/state_chain_observer/sc_observer/mod.rs`)

The same issue is present in the `TransactionBroadcastRequest` handlers for Polkadot and Bitcoin.

Recommendations

Short term, have the engine notify the State Chain if the Ethereum RPC API call fails by submitting a `transaction_signing_failure` extrinsic.

Long term, consider adding logic to the engine that attempts to resend the signed transaction.

18. Threshold signature liveness protection does not account for previously punished validators

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-CHFL-19
Target: `state-chain/pallets/cf-threshold-signature/src/lib.rs`	

Description

The Chainflip `cf-threshold-signature` pallet has an `offenders` function that decides which nodes should be reported for failure of the threshold signature ceremony. It does not punish more than half of the ceremony participants at one time, as this would leave insufficient validators for another ceremony attempt.

```
// The maximum number of offenders we are willing to report without risking the
// liveness of the network.
let liveness_threshold = self.participant_count / 2;
```

Figure 18.1: The liveness threshold is set to half the ceremony participants. (*`audit-chainflip/state-chain/pallets/cf-threshold-signature/src/lib.rs`*)

However, the implementation does not account for validators that have been punished in previous ceremonies. Therefore, half the ceremony participants (corresponding to one-third of the authority set) may be punished in one ceremony, and another half of the ceremony participants (which will be different validators because punished validators are suspended) may be punished in the next ceremony. Now, two-thirds of the authority set are suspended, so the protocol cannot progress.

Recommendations

Short term, base the liveness threshold on the number of suspended validators (or conversely, on the number of available validators in the authority set).

Long term, write negative tests that span multiple signing sessions for different protocol invariants.

19. A malicious minority can ruin liveness	
Severity: **Medium**	Difficulty: **Medium**
Type: Denial of Service	Finding ID: TOB-CHFL-20
Target: `engine/src/multisig/client/common/broadcast_verification.rs`	

Description

Signing ceremonies require a reliable broadcast, which currently consists of all participants sending their broadcast messages, as well as all broadcast messages that they receive, to all other participants. Then, only those messages that are confirmed by more than two-thirds of all participants are considered reliable. Any party that fails to reliably broadcast its message gets reported by all honest nodes.

```
// Check that the values are agreed on by the threshold majority.
// A party is reported if we can't agree on the value they broadcast
// or if the agreed upon value is `None` (i.e. they didn't broadcast)
for idx in &participating_idxs {
    let message_iter = verification_messages.values().map(|m| m.data[idx].clone());
    if let Some(Some(data)) = find_frequent_element(message_iter, threshold) {
        agreed_on_values.insert(*idx, data);
    } else {
        reported_parties.insert(*idx);
    }
}
```

Figure 19.1: Parties are reported if their values do not receive a threshold majority. (`engine/src/multisig/client/common/broadcast_verification.rs`)

For all ceremonies (except signing key verification ceremonies), the number of participants is only two-thirds of the full authority set. Therefore, for an authority set of 150 members, ceremonies will have 100 participants, so at least 67 parties must agree on a message.

This means that 34 malicious parties could collude to ensure that 50 other members of the authority set are punished in a certain ceremony. It is not possible to punish more than 50 members in a single ceremony due to the liveness protection mechanism (TOB-CHFL-19). Punishing 50 other members ensures that the malicious parties are selected for the repetition of the ceremony, where they can collude to ensure that another 50 members of the authority set are punished. This would prevent liveness of the system.

With 34 colluding validators, the probability that they are all pseudo-randomly selected for a ceremony is approximately 1 in 10 million. However, with 40 colluding validators, the probability that 34 or more of them are pseudo-randomly selected is approximately 3 in

1,000. With 50 colluding validators (which is the limit beyond which they can simply block any ceremony outright), this probability is 2 out of 3.

Because this issue does not lead to the loss of user funds, the severity is set to low. Because the required number of colluding validators is less than a third of the complete authority set, the difficulty is set to medium.

Exploit Scenario

Fifty validators in the authority set wait until at least 34 of them are selected for a signing ceremony. The colluding validators select 50 honest participants from the authority set and, during a broadcast, claim that they did not receive any messages from the selected parties. This causes all validators to report the 50 honest participants as malicious, which results in them being punished.

During the next ceremony, all colluding validators are automatically nominated to participate, and they repeat the process for another 50 honest participants. Now, no ceremonies can be conducted until the suspensions end.

Recommendations

Short term, increase the punishing threshold for a reliable broadcast to at least half of the participants. This will guarantee that a dishonest majority of between 34 and 50 participants cannot collude to punish other validators, and it will reduce the probability of enough colluding validators being nominated for the same ceremony. (With 50 colluding validators, the probability that they are all selected for a ceremony is approximately one in a trillion.)

Long term, consider documenting the maximum number of colluding validators that the overall system should withstand. This will make it possible to analyze the consequences of various thresholds.

20. Validators can report nonparticipants in ceremonies	
Severity: **Medium**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-CHFL-21
Target: `state-chain/pallets/cf-threshold-signature/src/lib.rs` and `state-chain/pallets/cf-vaults/src/lib.rs`	

Description

Participants in a threshold signing ceremony can report failure using the signed `report_signature_failed` extrinsic of the `cf-threshold-signature` pallet. They can provide a set of `ValidatorId` elements of offenders that should be punished according to the validator. The number of reports of offenders are tracked by the `context.blame_counts` field, which is another set.

```
for id in offenders {
    (*context.blame_counts.entry(id).or_default()) += 1;
}
```

Figure 20.1: Entries in the `blame_counts` set are created if they do not exist yet. (`state-chain/pallets/cf-threshold-signature/src/lib.rs`)

However, there is no check for whether the offenders were participants of the ceremony. In the case of a ceremony timeout, the offenders are punished according to the `offenders` function, which takes `context.blame_counts` and reports any element of the `BTreeSet` that was blamed by more than two-thirds of the ceremony participants, regardless of whether they were a participant of the ceremony.

```
let mut to_report = self
    .blame_counts
    .iter()
    .filter(|(_, count)| **count > blame_threshold)
    .map(|(id, _)| id)
    .cloned()
    .collect::<BTreeSet<_>>();
```

Figure 20.2: All validators that are blamed by at least two-thirds of the ceremony participants are reported. (`state-chain/pallets/cf-threshold-signature/src/lib.rs`)

The same issue holds for the `cf-vaults` pallet, where validators can report failures during the key generation ceremony:

```
for id in blamed {
        *self.blame_votes.entry(id).or_default() += 1
}
```

Figure 20.3: Entries in the `blame_votes` set are created if they do not exist yet.
(`state-chain/pallets/cf-vaults/src/lib.rs`)

In this case, all validators of the authority set are part of the ceremony. However, there are no checks for whether the `ValidatorId` corresponds to a member of the current authority set. Therefore, it would be possible to punish validators that are not part of the current authority set (e.g., back-up validators or historical validators).

Exploit Scenario

A group of 67 validators (in the `cf-threshold-signature` pallet) or 100 validators (in the `cf-vaults` pallet) colludes to punish validators that are not part of the ceremony, which causes the targeted validators to lose reputation or be suspended.

Because this does not lead to the loss of user funds, the severity is set to low. Because the attack requires more than a third of the authority set to collude, the difficulty is set to high.

Recommendations

Short term, add validation checks to the code to ensure that the reported participants are part of the ceremony.

Long term, write negative test cases to check that all functionality of the protocol rejects nonparticipating validators as inputs.

21. Staker funds can be locked via front-running

Severity: **High**	Difficulty: **Low**
Type: Denial of Service	Finding ID: TOB-CHFL-23
Target: `eth-contracts/contracts/StakeManager.sol`	

Description

The `stake` function in the `StakeManager` contract takes FLIP tokens from the caller to stake them in the protocol, ensuring that the amount is greater than the minimum staking amount defined. When the transfer is completed, the function emits a `Staked` event containing metadata about the operation. This event is then witnessed by the Chainflip engine and relayed to the `cf-staking` pallet on the State Chain.

```
function stake(
    bytes32 nodeID,
    uint256 amount,
    address returnAddr
) external override nzBytes32(nodeID) nzAddr(returnAddr) {
    IFLIP flip = _FLIP;
    require(address(flip) != address(0), "Staking: Flip not set");
    require(amount >= _minStake, "Staking: stake too small");
    // Assumption of set token allowance by the user
    flip.transferFrom(msg.sender, address(this), amount);
    emit Staked(nodeID, amount, msg.sender, returnAddr);
}
```

Figure 21.1: The FLIP `stake` function in the Ethereum network (`eth-contracts/contracts/StakeManager.sol`)

When this event is processed in the `cf-staking` pallet, the withdrawal address submitted is checked. If it is the first time a stake is made for that address and node ID, it goes through. However, if the node ID already has an address associated with it, this address must match the one submitted in the transaction; otherwise the staked funds are locked in the contract. This is because the Ethereum side already has the tokens, and only governance can recover them.

```
// If we reach here, the account already exists, so any provided withdrawal address
// *must* match the one that was added on the initial account-creating staking
event,
// otherwise this staking event cannot be processed.
match WithdrawalAddresses::<T>::get(account_id) {
   Some(existing) if withdrawal_address == existing => Ok(()),
   _ => {
      // The staking event was invalid - this should only happen if someone bypasses
      // our standard ethereum contract interfaces. We don't automatically refund
      // here otherwise it's attack vector (refunds require a broadcast, which is
      // expensive).
      //
      // Instead, we keep a record of the failed attempt so that we can potentially
      // investigate and / or consider refunding automatically or via governance.
      FailedStakeAttempts::<T>::append(account_id, (withdrawal_address, amount));
      Self::deposit_event(Event::FailedStakeAttempt(
         account_id.clone(),
         withdrawal_address,
         amount,
      ));
      Err(Error::<T>::WithdrawalAddressRestricted)
   },
}
```

Figure 21.2: Relevant part of the `check_withdrawal_address` function in the `cf-staking` pallet (`backend/state-chain/pallets/cf-staking/src/lib.rs`)

An attacker can front-run a valid staking transaction by submitting the same node ID as the pending transaction, the minimum staking amount, and a different return address. This locks the legitimate staker funds in the contract until governance releases them at the cost of the minimum staking amount.

If the node ID is known beforehand, there is no need to front-run because the stake can be sent any time before the legitimate transaction is sent.

Exploit Scenario

Mallory decides to prevent new validators from staking FLIP tokens. She sets up a monitoring system, and once she sees a pending stake transaction in the mempool, she front-runs it with another transaction with the same node ID.

Since the attack has virtually no cost for Mallory, she can keep doing this as long as she has enough FLIP tokens. This can effectively prevent any other party from interacting with the `stake` function and thereby prevent more validators from being added to the protocol.

Recommendations

Short term, consider removing the withdrawal address parameter and default it to be the caller. This will require checking integration with vesting contracts to make sure they can handle withdrawals.

Long term, consider using a double map structure that allows retrieving a withdrawal address given the staker and node ID keys.

22. Unbounded loop execution may result in out-of-gas errors

Severity: **Informational**	Difficulty: **Medium**
Type: Configuration	Finding ID: TOB-CHFL-24
Target: `contracts/Vault.sol`	

Description

The `Vault` contract has an externally callable `allBatch` function, which executes a series of deployments and transfers.

The `allBatch` function also calls a few other functions, including `deployAndFetchBatch`, `fetchBatch`, and `transferBatch`.

```
function allBatch(
    SigData calldata sigData,
    DeployFetchParams[] calldata deployFetchParamsArray,
    FetchParams[] calldata fetchParamsArray,
    TransferParams[] calldata transferParamsArray
)
    external
    override
    onlyNotSuspended
    consumesKeyNonce(
        sigData,
        keccak256(
            abi.encodeWithSelector(
                this.allBatch.selector,
                SigData(
                    sigData.keyManAddr,
                    sigData.chainID,
                    0,
                    0,
                    sigData.nonce,
                    address(0)
                ),
                deployFetchParamsArray,
                fetchParamsArray,
                transferParamsArray
            )
        )
    )
{
    // Fetch by deploying new deposits
    _deployAndFetchBatch(deployFetchParamsArray);
    // Fetch from already deployed deposits
    _fetchBatch(fetchParamsArray);

    // Send all transfers
    _transferBatch(transferParamsArray);
}
```

Figure 22.1: The `allBatch` function (`contracts/Vault.sol`)

The `deployAndTransferBatch` function, for example, deploys a new `Deposit` contract for each object in the `params` array. Because such logic is repeated for fetching and transferring funds, this function has the potential to run out of gas during execution.

```
function deployAndFetchBatch(
    SigData calldata sigData,
    DeployFetchParams[] calldata deployFetchParamsArray
)
    external
    override
    onlyNotSuspended
    consumesKeyNonce(
        sigData,
        keccak256(
            abi.encodeWithSelector(
                this.deployAndFetchBatch.selector,
                SigData(
                    sigData.keyManAddr,
                    sigData.chainID,
                    0,
                    0,
                    sigData.nonce,
                    address(0)
                ),
                deployFetchParamsArray
            )
        )
    )
{
    _deployAndFetchBatch(deployFetchParamsArray);
}

function _deployAndFetchBatch(DeployFetchParams[] calldata deployFetchParamsArray)
private {
    // Deploy deposit contracts
    uint256 length = deployFetchParamsArray.length;
    for (uint256 i = 0; i < length; ) {
        new Deposit{salt: deployFetchParamsArray[i].swapID}(
            IERC20Lite(deployFetchParamsArray[i].token)
        );
        unchecked {
            ++i;
        }
    }
}
```

Figure 22.2: The `deployAndFetchBatch` function (`contracts/Vault.sol`)

Recommendations

Short term, consider having the code run tests to estimate the gas consumption of each call, as a function of the `params` array's size, and make sure the amount of gas sent in the call is enough for the execution.

Long term, refactor the flow to remove the `Deposit` contract and simplify the handling of batch execution in the code, and consider using Echidna in "gas mode" to detect high gas use.

23. Anyone can cause the Chainflip engine to panic

Severity: **Medium**	Difficulty: **Low**
Type: Denial of Service	Finding ID: TOB-CHFL-25
Target: `engine/src/eth/vault.rs`	

Description

The Chainflip engine panics when decoding token values that exceed the maximum `u128` value. There is no extra validation in the Ethereum contracts, and as such, anyone can trigger a panic in the engine.

The `xSwapToken` function handles user tokens and emits an event for the engine to pick up.

```
function xSwapToken(
    uint32 dstChain,
    bytes memory dstAddress,
    uint16 dstToken,
    IERC20 srcToken,
    uint256 amount
) external override onlyNotSuspended nzUint(amount) {
    srcToken.safeTransferFrom(msg.sender, address(this), amount);
    emit SwapToken(dstChain, dstAddress, dstToken, address(srcToken), amount,
msg.sender);
}
```

Figure 23.1: The `xSwapToken` function (`contracts/Vault.sol`)

The event is handled by the `decode_log_closure` function in the Chainflip engine, which tries to decode the amount in the log and panics if it does not fit into a `u128`.

```
if event_signature == swap_token.signature {
    let log = swap_token.event.parse_log(raw_log)?;
     VaultEvent::SwapToken {
         destination_chain: utils::decode_log_param(&log, "dstChain")?,
         destination_address: utils::decode_log_param(&log, "dstAddress")?,
         destination_token: utils::decode_log_param(&log, "dstToken")?,
         source_token: utils::decode_log_param(&log, "srcToken")?,
         amount: utils::decode_log_param::<ethabi::Uint>(&log, "amount")?
             .try_into()
             .expect("SwapToken amount should fit into u128"),
         sender: utils::decode_log_param(&log, "sender")?,
     }
}
```

Figure 23.2: The relevant part of the `decode_log_closure` function (`engine/src/eth/vault.rs`)

The panic currently seems to be unhandled, which would likely cause the engine to halt.

For the supported tokens with real value, the assumption that the amount will never overflow is reasonable. However, since the Ethereum contracts accept any kind of token, these values can easily be exceeded through a malicious token.

This issue is also present in the `xCallToken` and `addGasToken` functions.

Exploit Scenario

Eve, an attacker, wants to compromise the Chainflip network. She deploys a dummy token and mints a supply greater than 2^128 - 1. She then calls `xSwapToken` with a large number of tokens and brings the Chainflip network to a halt due to the decoding error.

Recommendations

Short term, make sure the code handles edge cases where the number of tokens can overflow. An acceptable remedy could be to have the code first check for valid tokens before continuing to decode.

Long term, consider all edge cases that can cause panics and make sure these cannot be triggered maliciously.

24. Failed deposits are incorrectly witnessed as having succeeded

Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-CHFL-26
Target: `engine/src/eth/ingress_witnesser.rs`	

Description

Transactions for Ethereum deposits are not checked for their success status. As such, failed transactions are falsely credited to the user.

The ingress witnesser for Ethereum filters all transactions in a block that is sent to an ingress address.

```
async fn process_block(
    &mut self,
    epoch: &EpochStart<Ethereum>,
    block: &EthNumberBloom,
) -> anyhow::Result<()> {
    use crate::eth::rpc::EthRpcApi;
    use cf_primitives::chains::assets::eth;
    use pallet_cf_ingress_egress::IngressWitness;

    let txs = self.rpc.block_with_txs(block.block_number).await?.transactions;

     // ...

    let ingress_witnesses = txs
        .iter()
        .filter_map(|tx| {
            let to_addr = core_h160(tx.to?);
            if address_monitor.contains(&to_addr) {
                Some((tx, to_addr))
            } else {
                None
            }
        })
        .map(|(tx, to_addr)| IngressWitness {
            ingress_address: to_addr,
            asset: eth::Asset::Eth,
            amount: tx
                .value
                .try_into()
                .expect("Ingress witness transfer value should fit u128"),
            tx_id: core_h256(tx.hash),
        })
        .collect::<Vec<IngressWitness<Ethereum>>>();

    // ...

    Ok(())
}
```

Figure 24.1: Ethereum transactions are parsed by the witnesser.
(`engine/src/eth/ingress_witnesser.rs`)

The `self.rpc.block_with_txs` method internally uses the remote procedure call (RPC) `eth_getBlockByNumber` to obtain all transactions in a block. The returned transactions are processed without checking their receipts, which include the success status of a transaction (`0` for failed and 1 for success). Without checking the status, it is not possible to tell which transactions succeeded and which failed due to an error. A transaction containing value that is sent to the `Deposit` contract can fail in multiple ways, such as when it runs out of gas.

Ingress is completed once the deposit event has been witnessed and the `do_single_ingress` function is called in the `cf-ingress-egress` pallet.

```
fn do_single_ingress(
    ingress_address: TargetChainAccount<T, I>,
    asset: TargetChainAsset<T, I>,
    amount: TargetChainAmount<T, I>,
    tx_id: <T::TargetChain as ChainCrypto>::TransactionId,
) -> DispatchResult {
    let ingress = IntentIngressDetails::<T, I>::get(&ingress_address)
        .ok_or(Error::<T, I>::InvalidIntent)?;
    ensure!(ingress.ingress_asset ==
        asset, Error::<T, I>::IngressMismatchWithIntent);

    // Ingress is called by witnessers, so asset/chain combination should
    // always be valid.
    ScheduledEgressFetchOrTransfer::<T, I>::append(
        FetchOrTransfer::<T::TargetChain>::Fetch {
            intent_id: ingress.intent_id,
            asset,
        }
    );

    Self::deposit_event(Event::<T, I>::IngressFetchesScheduled {
        intent_id: ingress.intent_id,
        asset,
    });

    // ...

    match IntentActions::<T, I>::get(&ingress_address)
        .ok_or(Error::<T, I>::InvalidIntent)?
    {
        IntentAction::LiquidityProvision { lp_account } =>
            T::LpBalance::try_credit_account(
                &lp_account,
                asset.into(),
                amount.into()
            )?,
        // ...
    };

    T::IngressHandler::handle_ingress(
        tx_id.clone(),
        amount,
        ingress_address.clone(),
        asset
    );

    Self::deposit_event(Event::IngressCompleted {
        ingress_address,
        asset,
        amount,
        tx_id
    });
    Ok(())
}
```

Figure 24.2: The `lp_account` user account is credited once the event has been witnessed. (`state-chain/pallets/cf-ingress-egress/src/lib.rs`)

After the event has been witnessed, an action to fetch the funds is scheduled and the `lp_account` is credited immediately. The function does not check whether the ingress can complete successfully.

Exploit Scenario

Eve sends a transaction with 100 ether to the `Deposit` contract. Her transaction contains the payload to call the `fetch` function. This transaction immediately reverts, but she is still credited on the Chainflip State Chain.

Recommendations

Short term, to make the current process more robust, consider having the code perform the following actions:

- Check the transaction status in the transaction receipt.
- Check whether the address has received the funds.
- Emit a deposit event only if the `fetch` call can also be simulated successfully.

Long term, consider redesigning the ingress process for funds coming from Ethereum. If the funds are sent directly to the `Vault` contract, there should be no issue retrieving them, and events can log the incoming deposits. Events will be contained in a transaction's logs only if the transaction was successful.

25. Validators are not reimbursed for transactions submitted to external chains

Severity: **Low**	Difficulty: **Not Applicable**
Type: Data Validation	Finding ID: TOB-CHFL-27
Target: `state-chain/pallets/cf-broadcast/src/lib.rs`	

Description

The `cf-broadcast` pallet tracks the transaction fees incurred for successful broadcasts in the `signature_accepted` extrinsic using the `TransactionFeeDeficit` storage item.

```
TransactionFeeDeficit::<T, I>::mutate(signer_id, |fee_deficit| {
    *fee_deficit = fee_deficit.saturating_add(to_refund);
});
```

Figure 25.1: The transaction fee deficit for the signer is updated with the fee when the transaction has been submitted to the external chain. (`state-chain/pallets/cf-broadcast/src/lib.rs`)

However, this storage item is only updated, never read. This means that validators are never reimbursed for the fee incurred for the outgoing transaction.

Recommendations

Short term, have the code ensure that validators are credited for transactions to external chains.

Long term, clearly document currently unimplemented functionality to avoid potential reputational loss due to mismanaged user expectations.

26. MEV incentives are unclear and require further investigation

Severity: **Undetermined**	Difficulty: **Not Applicable**
Type: Undefined Behavior	Finding ID: TOB-CHFL-28
Target: `state-chain/pallets/cf-swapping/src/lib.rs`	

Description

The maximum extractable value (MEV) risks that a typical user is exposed to by actors aiming to extract as much value as possible are well studied in the case of Uniswap v3. The design of the automated market maker (AMM) stays close to the Uniswap v3 implementation but deviates in some important ways. These differences should be carefully considered, and the resulting economic and game theoretic incentives should be fully understood.

The following is a non-exhaustive list of noteworthy findings and differences from Uniswap v3:

- `tickSpacing = 1`
  - Uniswap's minimum tick spacing is 10 and can be as large as 2000.
  - While Rust's `BTreeMap` offers a more efficient way to query liquidity positions inside a certain range for range and limit orders, the effects of a malicious actor storing small amounts of liquidity that are all accessed when performing a swap have not been fully analyzed and could cause a potential denial of service (DoS) of the system.
- Two (non-stable) assets are always swapped through stable USDC.
- The network fee is always taken on the stable asset.
  - Through rounding, this could end up being zero.
  - Typically, a protocol fee is deducted from the input amount.
- Limit and range order funds are both accessed when performing a swap.
  - Output amount estimates might end up being harder to compute.
- The swap is aborted if there is insufficient liquidity.

    - Partial swaps are not allowed.
    - This could allow DoS attacks.
- Price limits are not enforced.
    - A limitation of batching swaps is that individual price limits are hard to enforce.
- There are no deadlines for swaps.
- Queued swaps are executed as a single, batched swap.
    - This lightens the computational load of the system.
    - This can create incentives that require further investigation.

We believe that the last four main bullet points particularly warrant further investigation and analysis because they could lead to new, unintended incentives for system participants. Slippage protection in a decentralized constant-product AMM like Uniswap is a core security feature that has been extensively analyzed. The effects of not including slippage protection and enforceable deadlines might mean that malicious actors can extract value from users' trades.

Since the Chainflip AMM batches swaps, front-running a swap is not possible, but value can still be extracted in much the same way as with ordered transactions. In fact, it might even be easier because, by batching transactions, one must only be included in the same batch to profit from another's swap. This widens the time window for actors in the system.

Exploit Scenario

A simplified scenario may illustrate this issue: Alice wants to swap $100 for ETH. Only range orders are being executed, there are no swap and network fees, Eve is the only other actor in the system, and liquidity is provided for the full price range, simplifying the AMM invariant to x * y = k.

In this setup, once Eve sees Alice's incoming swap, she decides to include a swap of $5000 in the same direction and then eventually back-run it. The swap output is divided by the proportions of the input, and since both transactions move the price further along the curve than if they had been executed individually, both receive a worse average buying price. However, Eve has effectively diminished Alice's returns. Eve's intention is to back-run the transaction (i.e., to perform a swap in the opposite direction with the output amount she received). Since there are no fees in the system in this scenario, it is easy to see that Eve will receive the funds that Alice lost in her swap.

This example is simplified, but it highlights the issue that is usually resolved by including slippage protection. Including swap fees will affect the outcome and potentially reduce Eve's profit. However, Eve can also be a liquidity provider herself, thereby reducing the impact again by being the one to receive the fees of her own swap. Furthermore, concentrated liquidity could further amplify this effect.

It is well known that manipulating prices for concentrated liquidity often requires less capital compared to providing the liquidity over the full range. This is because liquidity is often tightly centered around the fair asset price, and once the price is outside the bounds of concentrated liquidity, it becomes very sensitive to input amounts. Since the swaps do not have any price limits, there might be a new incentive to deploy capital at extreme prices. In Uniswap, there will not be an incentive to do so because swaps will never reach those limits due to slippage protection.

Consider the scenario of a sole liquidity provider with extensive funds who would like to maximize their profit. The optimal strategy might be to deploy a small amount of capital in the center (to give the illusion of a fair asset price) and at the far extremes of the possible price limits. That actor might wait for a few incoming swaps and then initiate large trades in both directions, effectively extracting the value of the queued swaps.

Another issue could arise from partial swaps not being executed. A single actor could stall swaps by initiating large trades that would use up all the remaining liquidity. The price could then be pushed in the opposite direction of the swap, and then sufficient liquidity could be provided to allow the queued swaps to be executed at an unfavorable price.

A similar case can be made for not allowing trade deadlines. If a swap can be stalled long enough, the price will be executed later, once it has potentially deviated far from its original position.

All these potential outcomes could cause price estimations to become very inaccurate, such as when liquidity is being deployed behind large gaps of little or no liquidity. The culmination of these effects could lead to a poor user experience, with bad actors diluting user profits by wash trading and deploying large amounts of liquidity at the far fringes of the price limits.

We acknowledge that these ideas have been presented in an ideal scenario where a sole malicious actor has full control over the entire liquidity provision and that they might not be economically feasible due to the opposing force that exists in the form of JIT liquidity. However, because there is no slippage protection for trades, there are no enforced deadlines, and the time window for malicious actors to act on swaps has widened, users could be exposed to higher risks.

Recommendations

Short term, include slippage protection and deadlines for swaps.

Long term, carefully analyze any noteworthy deviations from existing systems that could create new incentives. Consider game theoretic and economic incentives of the system that might arise and whether these are aligned with the protocol's goals.

A. Vulnerability Categories

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

Vulnerability Categories

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

Severity Levels

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

Difficulty Levels

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

B. Code Maturity Categories

The following tables describe the code maturity categories and rating criteria used in this document.

Code Maturity Categories	
Category	Description
Arithmetic	The proper use of mathematical operations and semantics
Auditing	The use of event auditing and logging to support monitoring
Authentication / Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Cryptography and Key Management	The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution
Decentralization	The presence of a decentralized governance structure for mitigating insider threats and managing risks posed by contract upgrades
Documentation	The presence of comprehensive and readable codebase documentation
Front-Running Resistance	The system's resistance to front-running attacks
Low-Level Manipulation	The justified use of inline assembly and low-level calls
Memory Safety and Error Handling	The presence of memory safety and robust error-handling mechanisms
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage

Rating Criteria

Rating	Description
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.
Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.
Not Applicable	The category is not applicable to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. Automated Testing

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

Clippy

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

Dylint

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

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

Figure C.1: Metadata required to run Dylint

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

Semgrep

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

We ran several custom Semgrep rules on the back-end repository. Because support for Rust in Semgrep is still experimental, we focused on identifying the following small set of issues:

- The use of panicking functions such as `assert`, `unreachable`, `unwrap`, and `expect` in production code (i.e., outside unit tests)

```
rules:
  - id: panic-in-function-returning-result
    patterns:
      - pattern-inside: |
          fn $FUNC(...) -> Result<$T> {
              ...
          }
      - pattern-either:
        - pattern: $EXPR.unwrap()
        - pattern: $EXPR.expect(...)
    message: |
      `expect` or `unwrap` called in function returning a `Result`.
    languages: [rust]
    severity: WARNING
```

Figure C.2: `panic-in-function-returning-result.yaml`

```
rules:
- id: unwrap-outside-test
  patterns:
  - pattern: $RESULT.unwrap()
  - pattern-not-inside: "
        #[test]
        fn $TEST() {
            ...
            $RESULT.unwrap()
            ...
        }
    "
  message: Calling `unwrap` outside unit test
  languages: [rust]
  severity: WARNING
```

Figure C.3: `unwrap-outside-test.yaml`

```
rules:
- id: expect-outside-test
  patterns:
  - pattern: $RESULT.expect(...)
  - pattern-not-inside: "
        #[test]
        fn $TEST() {
            ...
            $RESULT.expect(...)
            ...
        }
    "
  message: Calling `expect` outside unit test
  languages: [rust]
  severity: WARNING
```

Figure C.4: `expect-outside-test.yaml`

- The use of the as keyword in casting, which can silently truncate integers (e.g., casting `data.len()` to a u32 can truncate the input length on 64-bit systems)

```
rules:
- id: length-to-smaller-integer
  pattern-either:
```

```
  - pattern: $VAR.len() as u32
  - pattern: $VAR.len() as i32
  - pattern: $VAR.len() as u16
  - pattern: $VAR.len() as i16
  - pattern: $VAR.len() as u8
  - pattern: $VAR.len() as i8
  message: |
    Casting `usize` length to smaller integer size silently drops high bits
    on 64-bit platforms
  languages: [rust]
  severity: WARNING
```

Figure C.5: `length-to-smaller-integer.yaml`

- Unexpected comparisons before subtraction (e.g., ensuring that *x* is less than *y* before subtracting *y* from *x*), which may indicate errors in the code

```
rules:
- id: switched-underflow-guard
  pattern-either:
    - patterns:
        - pattern-inside: |
              if $Y > $X {
                  ...
              }
        - pattern-not-inside: |
              if $Y > $X {

              } else {
                  ...
              }
        - pattern: $X - $Y
    - patterns:
        - pattern-inside: |
              if $Y >= $X {
                  ...
              }
        - pattern-not-inside: |
              if $Y >= $X {

              } else {
                  ...
              }
        - pattern: $X - $Y
    - patterns:
        - pattern-inside: |
              if $Y < $X {
                  ...
              }
        - pattern-not-inside: |
              if $Y < $X {
              } else {
                  ...
              }
        - pattern: $Y - $X
    - patterns:
        - pattern-inside: |
              if $Y <= $X {
                  ...
              }
        - pattern-not-inside: |
              if $Y <= $X {

              } else {
                  ...
              }
        - pattern: $X - $Y
    - patterns:
        - pattern-inside: |
              if $Y > $X {

              } else {
                  ...
              }
        - pattern: $Y - $X
    - patterns:
        - pattern-inside: |
              if $Y >= $X {

              } else {
                  ...
              }
        - pattern: $Y - $X
    - patterns:
        - pattern-inside: |
              if $Y < $X {

              } else {
                  ...
              }
        - pattern: $X - $Y
    - patterns:
        - pattern-inside: |
              if $Y <= $X {

              } else {
                  ...
              }
        - pattern: $X - $Y
    - patterns:
        - pattern: |
              if $X < $Y {
              }
              ...
```

```
message: Potentially switched comparison in if-statement condition
languages: [rust]
severity: WARNING
```

Figure C.6: `switched-underflow-guard.yaml`

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-geiger

The `cargo-geiger` Cargo plugin provides statistics on the use of unsafe code in the project and its dependencies. The plugin can be installed using `cargo install cargo-geiger`. To run the tool, run `cargo geiger` in the crate root directory.

cargo-outdated

The `cargo-outdated` Cargo plugin identifies project dependencies with newer versions available. The plugin is installed by running `cargo install cargo-outdated`. To run the tool, run `cargo outdated` 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 command `cargo llvm-cov` in the crate root directory.

Echidna

Chainflip provided a base fuzzing test suite for the Ethereum smart contracts. We modified and extended this suite to ensure that all contract functions are covered by the fuzzing tests. The following are some of the changes we introduced:

- **Use of `allContracts` Echidna flag.** To extend coverage without adding function wrappers, the structure of the Echidna test file was refactored to use the real deployment script and contracts. This allows the use of `allContracts`, an Echidna flag that calls functions from all deployed contracts.

- **Addition of callback testing functionality.** Implementing a `CFReceiver` contract allows Echidna to fuzz and test message callbacks from cross-chain messages and to test for reentrancy issues.

- **Addition of message signing code.** Since the Chainflip contracts rely on the aggregate key signatures to validate messages, most privileged functions could not be fuzzed in the provided suite. Integration with a Python signing script was added, and messages can now be signed and pass signature validation.

- **Generation of new aggregate keys.** A Python script for generating valid signing keys was added to fuzz the key-changing functions in the `KeyManager` contract.

The original test suite implemented invariants and tested for reverts, but since no valid signature could be passed to the contracts' state-changing functions, most tests were static. We kept Chainflip's invariants but extended tests to make sure privileged functions were called correctly and that changes in the state of the contracts were persisted.

To test `Vault` callbacks, we implemented a new `CFReceiver` contract that also tests public functions for reentrancy. The code simulates an attacker contract trying to call random public functions from the protocol when a callback is received.

In addition, we added fuzzing tests for the FLIP ERC-20 token using our crytic/properties repository. These tests verify that the token implementation is compliant with the ERC-20 standard. We added the following tests:

- ERC-20 basic properties: tests for the basic functionality of ERC-20 tokens, such as approvals and transfers
- ERC-20 increase and decrease allowance properties: tests for the implementation of safe allowance handling
- ERC-20 mintable properties: tests for internal accounting and implementation of mintable tokens
- ERC-20 burnable properties: tests for internal accounting and implementation of burnable tokens

The Echidna parameters and token mock contract used for the fuzzing tests are provided below.

```
testMode: "assertion"
testLimit: 500000
shrinkLimit: 1500
codeSize: 0x60000
coverage: true
corpusDir: echidna-corpus/
allContracts: true
allowFFI: true
sender: ["0x3421b011C6a5f7c1ED438368565A5A8943063D19",
"0x6C377BdD40216c5b997Ab2bf0CFF8c2624F9Dd86", "0x10000", "0x20000", "0x30000"]
```

Figure C.7: Echidna configuration used for the extended test suite

}

For the ERC-20 property tests, we created a `CryticTokenMock` contract:

```
pragma solidity ^0.8.0;
import "../../FLIP.sol";
import {ITokenMock} from
"@crytic/properties/contracts/ERC20/external/util/ITokenMock.sol";
import {CryticERC20ExternalBasicProperties} from
"@crytic/properties/contracts/ERC20/external/properties/ERC20ExternalBasicProperties
.sol";
import {CryticERC20ExternalIncreaseAllowanceProperties} from
"@crytic/properties/contracts/ERC20/external/properties/ERC20ExternalIncreaseAllowan
ceProperties.sol";
import {CryticERC20ExternalMintableProperties} from
"@crytic/properties/contracts/ERC20/external/properties/ERC20ExternalMintablePropert
ies.sol";
import {CryticERC20ExternalBurnableProperties} from
"@crytic/properties/contracts/ERC20/external/properties/ERC20ExternalBurnablePropert
ies.sol";
import {PropertiesConstants} from
"@crytic/properties/contracts/util/PropertiesConstants.sol";


contract CryticERC20ExternalHarness is CryticERC20ExternalBasicProperties,
                                    CryticERC20ExternalIncreaseAllowanceProperties,
                                    CryticERC20ExternalMintableProperties,
                                    CryticERC20ExternalBurnableProperties {
    constructor() {
        // Deploy ERC20
        token = ITokenMock(address(new CryticTokenMock()));
    }
}

contract CryticTokenMock is FLIP, PropertiesConstants {

    bool public isMintableOrBurnable;
    uint256 public initialSupply;
    constructor () FLIP(
            10**30,
            1,
            10**18,
            address(0x10000),
            address(0x20000),
            address(0x10000)
        ) {
        _mint(USER1, INITIAL_BALANCE);
        _mint(USER2, INITIAL_BALANCE);
        _mint(USER3, INITIAL_BALANCE);
        _mint(msg.sender, INITIAL_BALANCE);

        initialSupply = totalSupply();
        isMintableOrBurnable = true;
    }
}
```

```
```

Figure C.8: The `CryticTokenMock` contract used for ERC-20 property testing

We used the following configuration for the ERC-20 property tests:

```
corpusDir: "echidna-corpus-erc20"
testMode: assertion
testLimit: 5000000
deployer: "0x10000"
sender: ["0x10000", "0x20000", "0x30000"]
allContracts: true
```

Figure C.9: The Echidna configuration used for ERC-20 property testing

D. Code Quality Recommendations

The following section contains code quality recommendations that do not have any immediate security implications.

- **Use fold rather than reduce in evaluate_polynomial** (in `engine/src/multisig/client/keygen/keygen_detail.rs`). This will avoid the call to `unwrap`.

```
coefficients
       .rev()
       .cloned()
       .reduce(|acc, coefficient| acc * Scalar::from(index) + coefficient)
       .unwrap()
```

- **Refactor the `StateChainClient::inner_new` method** (defined in `engine/src/state_chain_observer/client/mod.rs`). It consists of 300 lines of complex asynchronous code, which reduces readability.

- **Define the following value (in `Deposit.sol`) as a constant**. Using constants rather than hard coding values improves maintainability of the code.

```
if (address(token) == 0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE) {
```

- **Avoid code duplication in `Deposit.sol`.** Code duplication reduces maintainability because issues may be introduced if one instance is changed but not the other. The following code is an example of duplicate code.

```
if (address(token) == 0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE) {
    // solhint-disable-next-line avoid-low-level-calls
    (bool success, ) = msg.sender.call{value: address(this).balance}("");
    require(success);
} else {
    // Not checking the return value to avoid reverts for tokens with no
    // return value.
    token.transfer(msg.sender, token.balanceOf(address(this)));
}
```

- **Rename ETH_ZERO_ADDRESS.** The zero address should be renamed because staking to the zero address is not an expected flow. (Staking to the zero address would result in funds immediately being burnt.)

```
/// This address is used by the Ethereum contracts to indicate that no
/// withdrawal address was specified when staking.
```

```
///
/// Normally, this means that the staker staked via the 'normal' staking
/// contract flow. The presence of any other address indicates that the funds
/// were staked from the *vesting* contract and can only be withdrawn to the
/// specified address.
pub const ETH_ZERO_ADDRESS: EthereumAddress = [0xff; 20];
```

- **Break up long functions in the Chainflip engine.** Several functions in the Chainflip engine implementation are over 100 lines long, such as the following:
    - The `main` function in `engine/src/main.rs` is almost 250 lines long.
    - The `start` function in `engine/src/eth/witnessing.rs` is almost 200 lines long.
    - The `start` function in `engine/src/btc/witnesser.rs` is almost 130 lines long.

  This makes the code difficult to read and refactor.

- **Update documentation for the `set_blocks_for_epoch` and `force_rotation` extrinsics** (in `state-chain/pallets/cf-validator/src/lib.rs`). The comments for the two functions claim that the caller must be the root user, but the implementations both use `EnsureGovernance::ensure_origin` to verify the origin of the extrinsic.

- **Update documentation for the `new_cermony_attempt` function** (in `state-chain/pallets/cf-threshold-signature/src/lib.rs`). The comment claims that the function may return None, but the function does not return a value.

E. Risks of Third-Party Contract Calls

The `Vault` smart contract executes arbitrary third-party contract calls without validating their calldata. This practice could facilitate theft of user funds or cause undefined system behavior. We recommend that users review each third-party contract that can be called to verify that it behaves as expected. Users should ensure the following:

- **The contract is not upgradeable.** Upgradeable contracts introduce additional security risks, including the risk that an attacker could use the `create2` opcode to upgrade a contract to a malicious version.
- **The contract cannot self-destruct.** If a target contract can be destroyed, users may lose the funds they have deposited.
- **The contract's parameters are identified and documented.** Every argument and parameter must be documented and visible so that users can review them.
- **The contract uses immutable parameters rather than mutable ones whenever possible.** Immutable parameters reduce the risk of state manipulation by privileged users.
- **The contract uses `_msgSender()` instead of `msg.sender` to ensure proper OpenGSN support.** If the correct parameter is not used, `msg.sender` may be set to an incorrect address during a message call.
- **The contract has been reviewed by a third-party vendor.** Third-party reviews provide end users with additional assurance of a contract's security.

F. Incident Response Recommendations

This section provides recommendations on formulating an incident response plan.

- **Identify the parties (either specific people or roles) responsible for implementing the mitigations when an issue occurs (e.g., deploying smart contracts, pausing contracts, upgrading the front end, etc.).**
- **Document internal processes for addressing situations in which a deployed remedy does not work or introduces a new bug.**
    - Consider documenting a plan of action for handling failed remediations.
- **Clearly describe the intended contract deployment process.**
- **Outline the circumstances under which Chainflip will compensate users affected by an issue (if any).**
    - Issues that warrant compensation could include an individual or aggregate loss or a loss resulting from user error, a contract flaw, or a third-party contract flaw.
- **Document how the team plans to stay up to date on new issues that could affect the system; awareness of such issues will inform future development work and help the team secure the deployment toolchain and the external on-chain and off-chain services that the system relies on.**
    - Identify sources of vulnerability news for each language and component used in the system, and subscribe to updates from each source. Consider creating a private Discord channel in which a bot will post the latest vulnerability news; this will provide the team with a way to track all updates in one place. Lastly, consider assigning certain team members to track news about vulnerabilities in specific system components.
- **Determine when the team will seek assistance from external parties (e.g., auditors, affected users, other protocol developers) and how it will onboard them.**
    - Effective remediation of certain issues may require collaboration with external parties.
- **Define contract behavior that would be considered abnormal by off-chain monitoring solutions.**

It is best practice to perform periodic dry runs of scenarios outlined in the incident response plan to find omissions and opportunities for improvement and to develop “muscle memory.” Additionally, document the frequency with which the team should perform dry runs of various scenarios, and perform dry runs of more likely scenarios more regularly. Create a template to be filled out with descriptions of any necessary improvements after each dry run.

G. 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 June 26 to June 30, 2023, Trail of Bits reviewed the fixes and mitigations implemented by the Chainflip 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 27 issues described in this report, Chainflip has resolved 21, has partially resolved two, and has not resolved the remaining four. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Severity	Status
1	Step 2 of the handover protocol can be forged	Medium	Resolved
2	Hash function is used as KDF in handover protocol	Informational	Resolved
3	Ceremony participants can send many delayed messages	Informational	Resolved
4	Binding value can be zero	Informational	Resolved
5	The Chainflip back end and smart contracts have vulnerable dependencies	Medium	Resolved
6	Potential panic in KeyId::from_bytes	Informational	Resolved
7	Solidity compiler optimizations can be problematic	Undetermined	Unresolved
8	ERC-20 token transfer fails for certain tokens	High	Resolved
9	addGasNative is missing check for nonzero value	Informational	Resolved

10	StakeManager contains unnecessary receive function	Informational	Resolved
11	Missing events for important operations	Low	Resolved
12	Nonstandard ERC-20 tokens get stuck when depositing	High	Resolved
13	transfer can fail due to a fixed gas stipend	Informational	Resolved
14	Low number of block confirmations configured for external blockchains	Undetermined	Unresolved
15	Hard to diagnose error from default behavior during signer nomination	Informational	Resolved
16	Failed broadcast nominees are not punished if epoch ends during broadcast	Low	Resolved
17	Nominated broadcast signer does not always report failures in engine	Informational	Resolved
18	Threshold signature liveness protection does not account for previously punished validators	Informational	Unresolved
19	A malicious minority can ruin liveness	Medium	Resolved
20	Validators can report nonparticipants in ceremonies	Medium	Resolved
21	Staker funds can be locked via front-running	High	Resolved
22	Unbounded loop execution may result in out-of-gas errors	Informational	Resolved
23	Anyone can cause the Chainflip engine to panic	Medium	Resolved

24	Failed deposits are incorrectly witnessed as having succeeded	High	Resolved
25	Validators are not reimbursed for transactions submitted to external chains	Low	Unresolved
26	MEV incentives are unclear and require further investigation	Undetermined	Partially Resolved

Detailed Fix Review Results

TOB-CHFL-1: Step 2 of the handover protocol can be forged
Resolved. Chainflip now uses a different protocol for key handover, and the finding is no longer applicable.

TOB-CHFL-2: Hash function is used as KDF in handover protocol
Resolved. Chainflip now uses a different protocol for key handover, and the finding is no longer applicable.

TOB-CHFL-3: Ceremony participants can send many delayed messages
Resolved in PR #2992. The implementation of `CeremonyRunner::add_delayed` now ignores additional delayed messages.

TOB-CHFL-4: Binding value can be zero
Resolved in PR #3198. The implementation of `gen_rho_i` now checks whether the result from `P::Scalar::from_bytes_mod_order` is zero and explicitly sets the result to one in this case. This results in a slight bias in the distribution of the output from `gen_rho_i`, but it is extremely unlikely to cause any issues in practice.

TOB-CHFL-5: The Chainflip back end and smart contracts have vulnerable dependencies
Resolved. The vulnerable dependencies have been updated, and the Chainflip back-end repository now runs `cargo-audit` automatically as part of the CI pipeline. The dependency on OpenSSL has also been replaced by Rustls. OpenZeppelin has been updated to 4.8.3 in PR #395 and `minimist` has been updated to 1.2.6 in PR #443.

TOB-CHFL-6: Potential panic in KeyId::from_bytes
Resolved in PR #3201. The implementation now relies on the `Serialize` and `Deserialize` traits provided by Serde to serialize and deserialize key IDs.

TOB-CHFL-7: Solidity compiler optimizations can be problematic
Unresolved. As gas savings are important for Chainflip, they have decided to accept the risk involved with enabling the Solidity optimizer:

> *Almost all projects in blockchain use the optimizer for their code. We ran an analysis on bytecode size and especially gas consumption and concluded that given the importance of gas savings in our protocol, we should not turn the optimizer off.*

TOB-CHFL-8: ERC-20 token transfer fails for certain tokens
Resolved in PR #336. The `_transfer` function now checks the length of the return data before the value is decoded. The implementation now also includes tests for tokens with nonstandard behavior.

TOB-CHFL-9: addGasNative is missing check for nonzero value

Resolved in PR #336. The `addGasNative` method now uses the `nzUint` modifier to ensure that `msg.value` is nonzero.

TOB-CHFL-10: StakeManager contains unnecessary receive function
Resolved in PR #336. The `receive` function has been removed from the `StakeManager` contract.

TOB-CHFL-12: Missing events for important operations
Resolved in PR #336. The `govWithdrawNative` method has been removed from both the `StakeManager` and `KeyManager` contracts.

TOB-CHFL-13: Nonstandard ERC-20 tokens get stuck when depositing
Resolved in PR #336. The `transfer` function signature has been updated.

TOB-CHFL-14: transfer can fail due to a fixed gas stipend
Resolved in PR #336. The `govWithdrawNative` method has been removed from the `KeyManager` contract.

TOB-CHFL-15: Low number of block confirmations configured for external blockchains
Unresolved. The Chainflip team raised the number of required block confirmations for both Ethereum and Bitcoin. However, for proof of stake–based blockchains like Ethereum, it is not sufficient to wait for a fixed number of block confirmations. Rather, the implementation should accept transactions only from blocks that are considered finalized by the consensus algorithm.

TOB-CHFL-16: Hard to diagnose error from default behavior during signer nomination
Resolved in PR #3092. The `threshold_nomination_with_seed` function now returns None if `Validator::authority_count_at_epoch` returns None.

TOB-CHFL-17: Failed broadcast nominees are not punished if epoch ends during broadcast
Resolved in PR #3396. Failed broadcasters are now punished when broadcast storage is cleaned up, which ensures that failed broadcasters are punished even if the transaction has to be resigned.

TOB-CHFL-18: Nominated broadcast signer does not always report failures in engine
Resolved in PR #3412. The transaction broadcast implementation now reports failures to the State Chain for all supported external chains.

TOB-CHFL-19: Threshold signature liveness protection does not account for previously punished validators
Unresolved. This is an informational finding, and Chainflip decided not to implement the suggested fix for the following two reasons:

- *Having 1/3 of nodes suspended goes against our basic assumption that a minority of more than 1/3 cannot not work as expected (a malicious minority of 1/3 can anyways DDOS our systems as part of our design). Therefore, it is safe to assume that the number of suspensions at a time will be less than 1/3.*

- *The down side allowing more than 1/3 of the validators to get suspended is limited. All this would do is block the signing for at most 15 minutes (heartbeat interval) after which the nodes will start getting unsuspended and we will regain enough number of nodes to do the signing.*

TOB-CHFL-20: A malicious minority can ruin liveness
Resolved in PR #3128. The implementation of broadcast verification now requires only half the participants to agree on the broadcast message, which resolves the issue.

TOB-CHFL-21: Validators can report nonparticipants in ceremonies
Resolved in PR #3517. Both the `cf-vault` and `cf-threshold-signature` pallets now check that reported validators participated in the key generation or signing ceremony.

TOB-CHFL-23: Staker funds can be locked via front-running
Resolved in PR #3179. The State Chain now ignores the withdrawal address when funds are staked.

TOB-CHFL-24: Unbounded loop execution may result in out-of-gas errors
Resolved. The Chainflip team has run tests to estimate the amount of gas consumed by a single `allBatch` call as a function of the `params` array's size to ensure that the amount of gas sent is enough to cover the execution of the function.

TOB-CHFL-25: Anyone can cause the Chainflip engine to panic
Resolved in PR #3285. Events from the `Vault` contract are now handled uniformly by the `call_from_event` function, which returns an error if conversion fails.

TOB-CHFL-26: Failed deposits are incorrectly witnessed as having succeeded
Resolved in PR #3181. The engine now filters out failed transactions by checking the transaction status in the corresponding transaction receipt.

TOB-CHFL-27: Validators are not reimbursed for transactions submitted to external chains
Unresolved. This feature is currently not implemented. According to the Chainflip team, it will be available when the Chainflip Mainnet is launched.

TOB-CHFL-28: MEV incentives are unclear and require further investigation
Partially resolved. The Chainflip team has responded to the concerns outlined in the issue in their public documentation. However, Trail of Bits believes that the issue of MEV still

requires further investigation and attention due to the special conditions surrounding Chainflip's automated market maker (AMM).