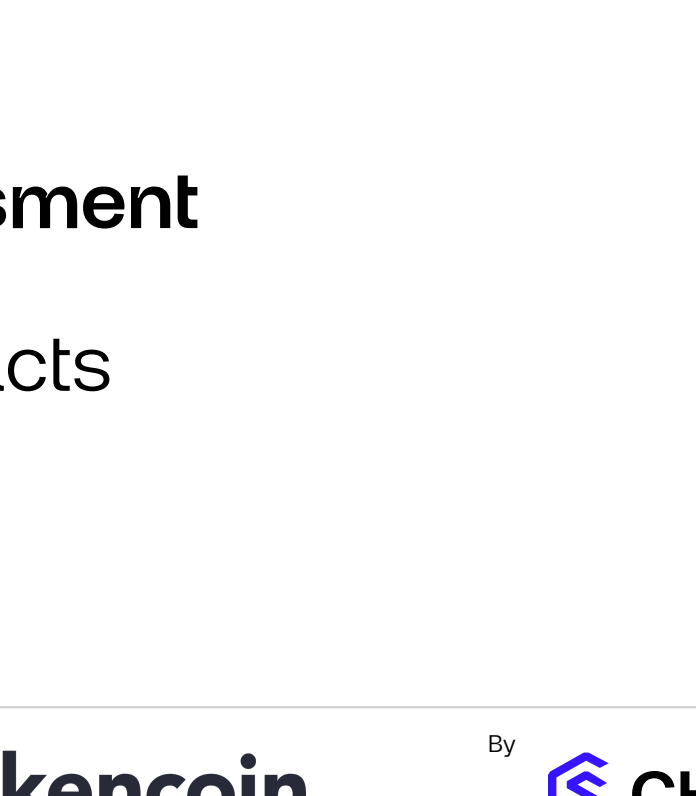
Code Assessment
of the FPS2 Smart Contracts

PUBLIC

Date July 14, 2026

Produced for  Frankencoin

By CHAINSECURITY



Contents

1 Executive Summary

Dear Frankencoin Team,

Thank you for trusting us to help Frankencoin with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of FPS2 according to **Scope** to support you in forming an opinion on their security risks.

Frankencoin implements FPS2, a wrapper that holds existing Frankencoin Pool Share tokens (FPS) one to one and adds a revised, binding governance layer on top, functioning as a shareholder agreement among participating FPS holders. Once more than two thirds of all FPS voting power has been wrapped into FPS2, the governance layer becomes binding and FPS2 holders gain the ability to permanently disempower FPS holders that have not joined. FPS2 implements the ERC-4626 interface with ZCHF as the underlying asset. A set of governance modules synchronizes FPS2 voting power to sidechains via CCIP, enabling cross-chain vetoes.

The most critical subjects covered in our audit are the integrity of the binding governance transition and the underlying vote accounting. The governance transition mechanism has been fixed by addressing **Shoot Enables Governance Takeover and Unlimited Minting**. The wrapping design was improved to address **Free Wrap/unwrap Round-Trips Can Delay FPS2 Binding Forever**. Security regarding the aforementioned subjects is now high.

Beyond the binding mechanism, we covered the cross-chain deployment of the governance modules and conformance with the ERC-4626 standard. We reviewed the ERC-4626 interface thoroughly, and found that several view and mutator functions deviate from the standard's requirements, see **ERC-4626 Spec Violations in FPS2MintRedeem** and **FPS2MintRedeem Mishandles Vault-Specific Limits, Violating ERC-4626**. After resolving these issues, compliance with 4626 spec is now good.

The general subjects covered are code quality, documentation, and functional correctness.

In summary, we find that the codebase provides a high level of security.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered, and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,
ChainSecurity

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the FPS2 repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

Version	Date	Commit Hash	Note
1	18 May 2026	f58abd7f7423268b4271c2921e1650da92473d6c	Initial review
2	22 June 2026	8ea0473baa3b71dd74c5b882d025f165893dd390	Fix review
3	14 July 2026	c1f229e3b26050367aafcb55da294342b4cae382	Second fixes

For the Solidity smart contracts, the compiler version `0.8.24` was chosen.

```
contracts/
├── bridge/
│   ├── CCIPAdmin.sol
│   └── ICCIPAdmin.sol
├── equity/
│   ├── Governance.sol
│   └── fps2/
│       ├── AccumulatingVotesToken.sol
│       ├── BridgedVotes.sol
│       ├── CCIPGovernance.sol
│       ├── FPS2.sol
│       ├── FPS2MintRedeem.sol
│       ├── GovernanceFactory.sol
│       ├── GovernanceModule.sol
│       ├── InterestGovernance.sol
│       ├── MainnetVotes.sol
│       └── MinterGovernance.sol
├── erc20/
│   └── CrossChainReference.sol
├── rate/
│   └── AbstractLeadrate.sol
├── stablecoin/
│   └── BridgedFrankencoin.sol
└── utils/
    └── MathUtil.sol
```

2.1.1 Excluded from scope

All contracts not explicitly listed above are out of scope.

3 System Overview

This system overview describes the most recent version (`Version 3`) of the contracts as defined in the Assessment Overview.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

Frankencoin implements the Frankencoin Pool Shares II (FPS2) smart contracts. FPS2 wraps the existing Frankencoin Pool Share tokens (FPS1) one to one and adds a governance layer with revised mechanisms on top, functioning as a shareholder agreement among participating FPS1 holders. The contract encodes a binding property: once a threshold of FPS1 voting power has been wrapped into FPS2, the governance layer becomes binding and FPS2 holders gain the ability to permanently disempower FPS1 holders that have not joined. In the following, we use the terms FPS1 and Equity interchangeably to refer to the existing Frankencoin shares system.

3.1 FPS2 (Frankencoin Pool Shares II)

The FPS2 contract is itself an ERC20 token and, from the perspective of the FPS1 contract, an ordinary FPS1 holder. Every FPS2 token is backed one to one by an FPS1 token held inside the contract.

FPS2 inherits two main components. AccumulatingVotesToken provides time-weighted vote accounting, largely adapted from the existing Equity contract. FPS2MintRedeem provides minting and redemption following the ERC-4626 tokenized vault standard. On top of this, FPS2 introduces a binding governance transition mechanism and a redemption discount that disincentivizes bank runs by making exits progressively more expensive. As redemption volume increases, remaining in the system becomes the rational choice, even if losses to the Equity are expected.

3.1.1 Transitioning from FPS to FPS2

Entry paths. Users can enter FPS2 in two ways. They can wrap existing FPS1 tokens, in which case `creditVotes()` carries over the votes the holder already had in FPS1. Alternatively, they can invest directly with ZCHF via the ERC-4626 `deposit` or `mint` functions, where the deposit is invested into FPS1 at the current FPS1 price. In this case, no votes are credited immediately.

Holders of WFPS are not credited votes when migrating. When they unwrap WFPS to FPS1 and then wrap into FPS2, they arrive with zero FPS1 votes and there is nothing to carry over.

Wrapping and unwrapping. Both `wrap()` and `unwrap()` convert one to one. `wrap()` pulls FPS1 into the contract and mints FPS2, then calls `creditVotes()` to carry over the FPS1 votes the sender lost on the transfer, converting the holder's accumulated governance weight from FPS1 votes to FPS2 votes. `unwrap()` burns FPS2 and returns FPS1, and is only available to holders whose holding duration exceeds the average holding duration across all holders.

Binding. FPS2 becomes binding when it controls more than two thirds of all FPS1 votes. The binding state activates the following mechanisms:

- **Shooting:** Once binding, anyone can call `shoot(target)` to destroy all FPS1 votes of a given FPS1 holder. Internally, the function calls `kamikaze` on the FPS1 contract with the target's current votes. The caller of `kamikaze` is the FPS2 contract, so FPS2 spends an equal amount of its own FPS1 votes as the destruction budget. Destroying a holder's FPS1 votes prevents them from participating in governance and from redeeming their FPS1 through the Equity contract for 90 days. The mechanism can be used to force remaining FPS1 holders into FPS2.

- **Reversibility:** The binding state is not permanent. If the voting power of FPS2 is reduced, e.g. through unwrapping, redemptions, or enough new FPS1 minted outside the contract, the vote share can fall back below two thirds, at which point `unwrap()` becomes available again and `shoot()` is no longer callable.

Vote accounting. Most functions in FPS2 and AccumulatingVotesToken are adapted from the existing Equity contract. Votes grow linearly with both balance and holding time. The `attack()` function (renamed from `kamikaze` in FPS1) lets a holder sacrifice their own votes to destroy an equal number of votes from other addresses. Delegation through `delegateVoteTo()` is non-subtractive and transitive. Each address can only appear once in a delegation chain, preventing double-counting of votes.

Vote capping. New compared to FPS1 is the permissionless `cap()` function. If a holder's `holdingDuration()` exceeds `HOLDING_DURATION_CAP` (365 days), the function moves that holder's anchor to one year ago, reducing both the holder's votes and `totalVotes` by the same amount. This prevents addresses from accumulating excessive voting power by having very long holding durations. The reduction of `totalVotes` is selective: it occurs only when `cap()` is explicitly applied to a specific holder.

3.1.2 Minting and Redemption

Redemption discount. FPS2 removes the 90-day holding-period requirement enforced by the existing Equity contract. Instead, FPS2 can be redeemed while the contract is binding, provided that the FPS2 contract's average holding duration in the FPS1 system exceeds 90 days. The redemption price (the bid) is reduced by a discount that depends on how much FPS2 was recently redeemed. The contract tracks recent redemptions in `weightedRecentRedemptions()`, which decays linearly to zero over `RECOVERY_PERIOD` (7 days). The discount factor is `discount = ((supply - shares/2) / (supply + recentRedemptions))^4`, computed via `_power4`. The fourth-power curve means small redemptions in calm markets face only a small discount, while large or rapid redemptions incur steep discounts. As redemption volume grows, the discount widens, making each successive exit less attractive than the previous one. The proceeds withheld by the discount are not distributed to the redeemer but remain in the Equity contract, increasing the value backing the remaining holders' shares.

ERC-4626 interface. FPS2 implements the ERC-4626 tokenized vault standard, with ZCHF as the underlying asset and FPS2 as the share token. The standard entry and exit functions (`deposit`, `mint`, `withdraw`, `redeem`) are provided, together with the corresponding preview and limit views. The ask price for mints and deposits equals `FPS1.price()`. The bid price for redeems and withdrawals equals `FPS1.price()` multiplied by the current discount. Because `mint` and `withdraw` fix the output amount, the required input is found by inverting the FPS1 pricing curve. A single `withdraw` may burn at most 10% of the total FPS2 supply, and larger requests revert. The share-denominated `redeem` path is not subject to this cap. FPS2 also provides frontrun-protection variants such as `redeemExpected`, which redeems only if the effective proceeds meet a caller-supplied minimum.

3.2 FPS2 Governance

The FPS2 contract cannot take part in governance directly. Instead, a set of governance contracts expose governance actions to qualified FPS2 holders by wrapping the underlying FPS1 governance logic with an FPS2-specific quorum check. The `onlyQualified` modifier gates all governance actions and requires the caller to hold more than 1% of the total FPS2 voting power (the `QUORUM` constant for FPS2 is 100 basis points).

Dual quorum requirement. Since the governance contracts ultimately invoke governance functions on the FPS1 layer, two thresholds must be met simultaneously: the caller must hold more than 1% of the voting power within FPS2, and the FPS2 contract itself must hold at least 2% of the voting power in FPS1. During the early transition period, as soon as FPS2 has surpassed the 2% mark in FPS1, the effective minimum to perform governance actions from within FPS2 is as low as 1% of 2% of total FPS1 votes.

Deployment. The governance contracts are wired up by GovernanceFactory, a deployment-only contract called from the FPS2 constructor. It deploys the appropriate set of governance contracts (MainnetVotes or BridgedVotes, CCIPGovernance, MinterGovernance, InterestGovernance) and delegates the FPS2 contract's FPS1 votes to the resulting governance contract. This delegation chain is what allows the governance

contracts to exercise FPS1-level governance on behalf of FPS2. The governance contracts are deployed to the same addresses on all chains, which allows the delegation chain to also confer voting power to the non-mainnet contracts. For more details on this, see FPS2 Governance Address DoS

The following governance contracts are deployed:

- **MinterGovernance** (mainnet and sidechains): governs adding and vetoing minters. `suggestMinter` enforces a minimum application period of 60 days and a minimum application fee of 1200 ZCHF. `denyMinter` and `denyPosition` are gated by `onlyQualified`. `denyUnannouncedMinter` is permissionless and incentivized: it pays the caller 10% of the reward pool for vetoing a minter proposal that was submitted directly to the legacy FPS1 governance, bypassing FPS2. This guards against minter proposals that circumvent the new FPS2 minimum application period.
- **CCIPGovernance** (mainnet and sidechains): exposes CCIP admin governance to qualified FPS2 holders. It wraps `proposeRemotePoolUpdate`, `proposeAddChain`, `proposeRemoveChain` and `proposeAdminTransfer`, as well as `applyRateLimit` and `deny` in single and batched variants. All functions delegate to the CCIP admin, passing the FPS2 contract as the helper to pass FPS1 quorum.
- **InterestGovernance** (mainnet only): lets qualified FPS2 holders propose adjustments to the system interest rates. `proposeBorrowingRate` and `proposeSavingsRate` take a rate in parts per million and delegate to the respective Leadrate contracts, which apply a 7-day grace period before the new rate takes effect. Sidechains have no local proposal path. Rates are propagated from mainnet to sidechains by LeadrateSender via CCIP to a BridgedLeadrate contract on each target chain, which updates the local rate. The sidechain side only receives and applies rates, it cannot propose them.

3.2.1 Bridging FPS2 Votes

FPS2 tokens are only minted on mainnet and are not bridged within the Frankencoin system. To enable governance participation on sidechains, a representation of FPS2 voting power is bridged via CCIP.

On mainnet, MainnetVotes serves as the canonical source of FPS2 voting state. On each sidechain, a BridgedVotes contract, deployed to the same address as MainnetVotes, receives and stores the bridged voting data. Both contracts inherit from Governance, which provides the delegation mapping and the quorum checks used by the governance contracts. The CCIP setup and trust assumptions are as in the previous CCIP Bridge report.

Syncing. The sync is unidirectional, from mainnet to sidechains. MainnetVotes acts as a CCIP sender, BridgedVotes as a CCIP receiver. Anyone can trigger a sync by calling the send function on MainnetVotes with a user-specified set of voter addresses and a target chain, provided they pay the required CCIP fee (payable in the native asset or LINK). The message payload is fixed: it contains each voter's current votes and delegate, plus the current `totalVotes`. Users may supply arbitrary `extraArgs` to configure CCIP execution parameters such as gas limits and out-of-order execution, but cannot pass an arbitrary payload. When the message arrives at the destination chain, the CCIP Router calls BridgedVotes, which updates its stored voting state accordingly.

Cross-chain pre-conditions. Two steps are required before an FPS2 holder can participate in governance on a sidechain. First, the FPS1 votes of the FPS2 contract itself must be synchronized to the target chain via GovernanceSender's `pushVotes`. Second, the individual FPS2 holder's votes must be synced via MainnetVotes' `pushFPS2Votes`.

Invariant divergence on sidechains. On mainnet, the invariant $\mathrm{totalVotes} = \text{sum of all individual votes}$ holds, because both values are derived from the same anchor state. On sidechains this invariant does not hold, as `totalVotes` and individual vote balances are independent snapshots that may have been synced at different times. Three factors contribute to this divergence:

1. Selective syncing: only the addresses included in a given sync message are updated, while `totalVotes` is updated on every sync. Interactions with `attack()` on mainnet between syncs can further widen the gap (see Effective FPS2 Governance Quorum Can Be Temporarily Lowered system consideration).

2. Vote capping on mainnet: `cap()` reduces `totalVotes` but only reduces individual votes when explicitly applied to a specific holder. A sync after a cap can propagate the reduced total without the corresponding individual reduction.
3. Local delegation: delegations can be created directly on a sidechain and are independent of the mainnet delegation state. A subsequent sync from mainnet overwrites any local delegation for the synced addresses (see Asynchronous Delegations system consideration).

3.3 Changes in Version 2

The following changes were made in **Version 2** of the contracts:

- FPS2 is now considered binding once it holds 2/3 of the total FPS1 votes. Previously it was binding at 1/2.
- `withdraw` and `redeem` in FPS2 are now only possible when binding.
- Unwrapping FPS2 into FPS1 can now only be done if the user has a holding duration that is greater than the average holding duration over all users. This prevents misusing the wrap/unwrap mechanism to destroy the accumulated votes of FPS2.
- The `MIN_APPLICATION_FEE` for minters in MinterGovernance was increased from 1000 to 1200 ZCHF. The amount that is used to refill the reward pool is now at most 200.
- In FPS2MintRedeem, the `convertToAssets` function no longer reflects the temporary discount based on recent redemptions.
- The overloaded `deposit` function was renamed to `depositExpected` and now takes a `recipient` parameter.

3.4 Changes in Version 3

The following changes were made in **Version 3** of the contracts:

- The `FPS2.unwrap()` function is now allowed even when FPS2 is binding. The above-average holding duration requirement still applies.

4 Trust Model

The FPS2 contracts do not modify the existing Frankencoin, Equity, MintingHub, Savings, or CCIP Bridge contracts. The trust assumptions established in the previous code assessments (Frankencoin, Frankencoin v2024, and the CCIP Bridge) continue to apply in full.

No new privileged roles are introduced. All governance actions within FPS2 require a qualifying quorum of 1% of total FPS2 voting power, compared to the 2% threshold in FPS1. Any qualified FPS2 holder can veto new minters, propose or veto CCIP configuration changes, and propose interest rate updates.

We assume that there is no malicious group of FPS2 holders that collectively have more than 50% of the FPS2 voting power after FPS2 has become binding. FPS2 holders are trusted to continuously monitor and veto proposals for adding untrusted minters, bad CCIP configuration changes, and interest rate changes that are too large.

The underlying FPS1 (Equity) contract and the ZCHF (Frankencoin) contract are assumed to behave as specified in the previous assessments. FPS2 relies on the correctness of the FPS1 pricing curve, vote accumulation, holding-duration enforcement, and redemption logic.

For cross-chain vote synchronization, FPS2 relies on the Chainlink CCIP infrastructure. The same trust assumptions as in the CCIP Bridge code assessment apply. If CCIP can relay incorrect values, Governance on that chain is fully compromised, which can lead to minting unlimited ZCHF on that chain.

Governance decisions on sidechains are based on vote snapshots that may be stale. The system depends on timely and sufficiently complete vote syncs for sidechain governance to function correctly.

The BridgedVotes contract validates incoming CCIP messages by checking that the sender address matches its own address, relying on the assumption that MainnetVotes and BridgedVotes are deployed at the same address across chains. This is enforced by the GovernanceFactory and Arachnid CREATE2 deployment mechanism.

Once binding, anyone can call `shoot()` to permanently destroy the voting power and redemption rights of any FPS1 holder that has not joined the wrapper. This is the intended enforcement mechanism to consolidate governance into FPS2.

The redemption discount is assumed to be parameterized sufficiently to deter strategic mass redemptions ahead of anticipated losses.

The deployment of the FPS2 governance contracts is out of the scope of this review and is trusted to be non-malicious and correctly configured. On mainnet, deployment is atomic via the FPS2 constructor. On sidechains, the `GovernanceFactory.deploy()` call is permissionless and is assumed to be performed before any adversary can front-run it with an incorrect `fps2mainnet` address.

5 System Considerations

We leverage this section to highlight findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require a modification inside the project. Instead, they should raise awareness in order to improve the overall understanding for users and developers.

SC1 Asynchronous Delegations

System Consideration | Version 1

Votes are strictly synced from mainnet to other chains unidirectionally. While delegations are synced in the same way, they can also be created on the L2s directly.

Users creating delegations on L2 should be aware that anyone can overwrite their delegations at any time if they are not in sync with mainnet.

This system consideration has been carried over from the Frankencoin CCIP Bridge report.

SC2 Effective FPS2 Governance Quorum Can Be Temporarily Lowered

System Consideration | Version 1

User A can `attack()` someone else using their full voting power x, then bridge only the victim's voting power. This would lead to the snapshot on the sidechain being inconsistent with A's voting power still being x, while the total vote count is $T - 2x$ (T being the old value for the vote count). For x to be above 1% of $T - 2x$, it is sufficient for x to be 0.980% of T. Therefore, this attack temporarily lowers the governance threshold to reach a qualifying quorum. The threshold can be restored again, however, by anyone bridging A's voting power.

A less-effective version of this attack involves user A calling `redeem()` and then bridging somebody else's voting power. The snapshot on the sidechain then has x as A's votes, and $T - x$ as the total votes count.

This system consideration has been carried over from the Frankencoin CCIP Bridge report and adjusted accordingly.

SC3 Governance Discrepancy in CCIPGovernance

System Consideration | Version 1

`CCIPGovernance` is FPS2 quorum-gated and calls into `CCIPAdmin` to execute governance functions. `CCIPAdmin` enforces a 7-day waiting period for the removal of a chain. However, setting the rate limit of the same chain to 0 does not trigger a waiting period and has a similar effect. Anyone who passes the quorum can also immediately increase the rate limit.

This system consideration has been carried over from the Frankencoin CCIP Bridge report and adjusted accordingly.

SC4 Migration to FPS2 Should Be Gradual

System Consideration | Version 1

In order for FPS2 to execute governance functions, it must have at least 2% of all FPS voting power. As voting power is the product of balance and average time held, the voting power of FPS2 will initially be low, as it does not have a lot of time accumulated. This means it will be possible for FPS1 holders who have a much lower balance but a lot of accumulated time to use the `kamikaze()` function to bring the FPS2 votes from above 2% to below 2%.

To protect from this, FPS1 holders with a large amount of voting power should move into FPS2 gradually. This allows them to retain the time they have accumulated on a part of their balance, which they can use to veto even in case FPS2 becomes the target of `kamikaze()` from a minority.

SC5 Users Should Split Very Large FPS2 Redemptions

System Consideration | Version 1

In FPS2MintRedeem, `discount()` prices the whole redemption at a single midpoint of a convex 4th-power curve:

```
uint256 leftMiddle = currentSupply - plannedRedemption / 2;
uint256 factor = _divD18(leftMiddle, total);
return _power4(factor);
```

Because the curve is convex, the midpoint value is below the true average over the redeemed range. Redeeming in small pieces realizes that higher average instead.

Redeeming 20% of total supply as two 10% pieces instead of one yields ~1.9% more. The effect compounds when redemptions stack in a short time: after 20% of supply has already been redeemed this block, an additional 20% split into two 10% pieces yields ~3.1% more compared to redeeming the same amount in a single call.

Users redeeming a large percentage of the total FPS2 supply at once should split their redemptions into two or more function calls.

6 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- **Likelihood** represents the likelihood of a finding to be triggered or exploited in practice
- **Impact** specifies the technical and business-related consequences of a finding
- **Severity** is derived based on the likelihood and the impact

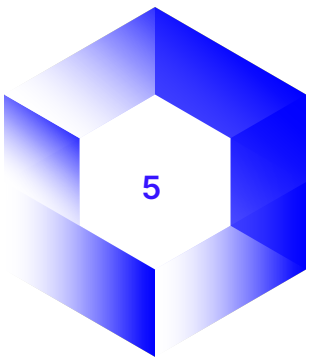
We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	**Impact**		
	High	Medium	Low
High	**Critical**	**High**	**Medium**
Medium	**High**	**Medium**	**Low**
Low	**Medium**	**Low**	**Low**

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

7 Findings

In this section, we describe the findings identified during the course of the engagement. The findings are categorized by severity as explained in the Terminology section. Below we provide an overview of the total number of findings per severity, as well as the number of open issues.



Open Issues

Total number of findings

Critical High Medium Low Informational

7.1 Overview

The following table lists all identified findings along with their severity and current status. A finding is considered resolved once it has been fully corrected or the specification has been changed accordingly. Non-informational findings with any other status, including partial fixes, acknowledgements, and accepted risks, remain open.

Critical	#001	Shoot Enables Governance Takeover and Unlimited Minting	Code Corrected
High	#002	Discount Computation Uses Incorrect totalSupply	Code Corrected
High	#003	FPS2 Governance Address DoS	Specification Changed
High	#004	Free Wrap/unwrap Round-Trips Can Delay FPS2 Binding Forever	Specification Changed
Medium	#005	ERC-4626 Spec Violations in FPS2MintRedeem	Code Part. Corrected / Acknowledged
Medium	#021	FPS2MintRedeem Mishandles Vault-Specific Limits, Violating ERC-4626	Code Part. Corrected / Risk Accepted
Medium	#022	Inverted Holding-Duration Check in Unwrap	Code Corrected
Medium	#023	maxRedeem Returns an Amount Instead of a Share Count	Code Corrected
Low	#006	Arbitrarily Old Vote Snapshots Can Be Applied	Risk Accepted
Low	#007	Binary Search Assumes Monotonicity of Effective Proceeds	Code Corrected
Low	#008	GovernanceFactory Does Not Implement Correct Interface, Blocking FPS2 Deployment	Code Corrected
Low	#009	MutualDestruction Event Reports Cumulative Votes Instead of the per-Target Amount	Code Corrected
Low	#010	Redemption Discount Recovers Slower than Intended	Specification Changed
Low	#011	totalAssets Counts Unredeemable FPS1	Specification Changed
Low	#012	Very Large Redemptions Can Reach Negative Marginal Price	Risk Accepted
Low	#024	`totalAssets()` Reverts Once FPS2 Owns Nearly the Entire FPS1 Supply	Specification Changed
Low	#028	`denyUnannouncedMinter` Can Be Disabled	Code Part. Corrected / Risk Accepted
Info	#013	`shoot()` Does Not Explicitly Reject FPS2 as Target	Code Corrected

Info	#014	Abstract Leadrate Does Not Inherit ILeadrate	Code Corrected
Info	#015	Suggesting a Minter Reverts Despite Paying the Contract's Stated Minimum Fee	Specification Changed
Info	#016	Inconsistent Function Naming	Code Corrected
Info	#017	Non-Indexed Event Parameters	Code Corrected
Info	#018	Outdated and Incorrect Documentation Comments	Code Corrected
Info	#019	Unused Error	Code Corrected
Info	#020	Unused Imports	Code Corrected
Info	#025	CCIP Rate Limit Increases Could Be Delayed	Acknowledged
Info	#026	previewMint of Zero Shares Returns a Positive ZCHF Amount	Acknowledged
Info	#027	Unused Variable in MainnetVotes	Code Corrected

7.2 Descriptions

#001 Shoot Enables Governance Takeover and Unlimited Minting

Design | Critical | Version 1 | Code Corrected

FPS2 introduces the `shoot` function, which allows anyone to use the `FPS1.kamikaze` function to cancel votes of FPS2 and another target address. This is intended to be used to enforce the transition to FPS2 once it has reached more than 50% of all FPS1 votes. If it is used to shoot all remaining FPS1 holders equally, this achieves the goal of transitioning fully to FPS2.

However, if an attacker shoots all other FPS1 holders aside from themselves, this can allow them to fully take over governance for some time, allowing them to mint unlimited unbacked Frankencoins.

Consider the following example, with an Attacker A, honest FPS1 holders B and C, and FPS2. The table shows absolute vote weights per actor at each step. Read each value as `* k` for some scaling factor `k` (e.g. `k = 1e27`). `ε` denotes an infinitesimal excess used to keep FPS2 strictly above the `> 50%` binding threshold.

Step	A	B	C	FPS2	Total
1	3	27	20	50 + ε	100 + ε
2	3	27	0	30 + ε	60 + ε
3	3	0	0	3 + ε	6 + ε
4	3	0	0	3	6
5	3	0	0	0	3

The same example, expressed as percentages of the total FPS1 votes. `ε` here differs between rows, but stays tiny throughout.

Step	A	B	C	FPS2
1	3% - ε	27%	20%	50% + ε
2	5% - ε	45%	0%	50% + ε
3	50% - ε	0%	0%	50% + ε
4	50%	0%	0%	50%
5	100%	0%	0%	0%

Steps:

1. **Initial state.** FPS2 just cleared the binding threshold (`FPS1.relativeVotes(FPS2) > 50%`), enabling `shoot`.
2. **A calls `shoot(C)`.** This destroys equal vote weights from both FPS2 and C. C is wiped to 0. FPS2 loses 20. Because the total drops by `2×` what FPS2 loses, FPS2's share stays just above 50% and remains binding.
3. **A calls `shoot(B)`.** B is wiped to 0 and FPS2 loses an equal amount of votes. FPS2's share is still just above 50%. This can be repeated for an arbitrary number of other FPS1 holders, leaving only A and FPS2 with votes.
4. **A mints/redeems FPS2 to drop it below binding.** Using some additional ZCHF, A mints FPS2, then immediately redeems it. FPS2's accumulated votes drop proportionally to the balance reduction. A sizes the

redemption to push FPS2 from `50% + ε` down to exactly `50%`. `isBinding()` now returns false, unlocking `unwrap`.

5. **A drains FPS2's votes to zero via repeated wrap/unwrap.** Each `wrap`/`unwrap` cycle reduces FPS2's accumulated votes. With enough cycles, FPS2's votes go to 0 as each `wrap` advances the vote anchor in time while each `unwrap` keeps the anchor while reducing FPS2's balance. Finally, A holds 100% of the remaining votes. While step 4 had a cost (deposit/redeem fee), this step is free.

6. **A creates a bad position that is impossible to veto.** With no remaining FPS1 holder able to reach the veto threshold, A submits a position proposal with a bad collateral that only they have access to. They periodically use `kamikaze` on all other addresses to ensure they do not collectively reach 2% of votes. After 3 days, the position becomes active and A can mint an unlimited amount of unbacked ZCHF.

Requirements:

The attacker must initially hold votes that are more than the total supply of FPS1 multiplied by the number of seconds in 3 days, as this is what they need for the `kamikaze` in step 6 (with a safety buffer in case others mint more FPS1 during the 3 days). FPS2 must initially have votes just barely above 50% (deposit/redeem can be used to bring the initial value down to the ideal, in case it is too large).

Costs:

Aside from gas costs, only step 4 has a cost (percentage fee on `deposit`/`redeem`). Additionally, there is the cost of capital required to achieve the necessary initial votes and the amounts needed to `wrap/unwrap` (using larger amounts requires fewer cycles/less gas).

> **Code Corrected**
>
> The `binding` level has been increased from 1/2 to 2/3. This means a `shoot` now increases the percentage of total votes that FPS2 controls. The exploit relied on shooting multiple times while staying at a consistent vote percentage, which is only possible at 50%. Additionally, unwrapping now requires having a holding duration that is greater than that of the average FPS2 user. This prevents wrap/unwrap loops within a short amount of time. Additionally, redeeming through FPS2 can now only be done while `binding`.
>
> The combination of these changes makes the described attack infeasible.

#002 Discount Computation Uses Incorrect totalSupply

Correctness	High	Version 1	Code Corrected

In `_redeem`, `_burn` is executed before `calculateEffectiveProceeds` is called. This leads to an incorrect `discount` computation. Because the burn already reduced `totalSupply`, the discount applied at redemption is computed against the post-burn supply rather than the pre-burn supply that the view functions and binary searches use.

```
uint256 currentSupply = totalSupply();
uint256 total = currentSupply + recentRedemptions;
uint256 leftMiddle = currentSupply - plannedRedemption / 2;
```

Consequences:

1. **The applied discount is wrong.** The discount is meant to be computed against the supply present at redemption. Reducing `totalSupply` before the computation lowers the discount factor, so the redeemer receives less ZCHF than the intended pricing curve specifies.

2. **View/execution mismatch.** `previewRedeem`, `maxWithdraw`, `currentDiscount`, and `bid` compute the discount on the pre-burn supply, while the executed redemption uses the post-burn supply. The discount

does not behave as a pure function of supply, so results diverge from the previews. In particular, `previewRedeem` returns a higher value than the proceeds actually paid by `redeem`. This is an ERC-4626 spec violation.

3. **Binary search uses an incorrect discount.** `_findSharesForAssets` evaluates `discount` at the current (pre-burn) `totalSupply`. The real redemption burns first, so the discount differs from the one the search optimized against. `withdraw` can therefore return fewer shares than required to obtain `assets`, sending the user less than `assets`. This is an ERC-4626 spec violation.

4. **Underflow reverts on large redemptions.** `leftMiddle = currentSupply - plannedRedemption / 2` underflows when `plannedRedemption > 2 * currentSupply`. Since the burn reduces `currentSupply` first, redemptions large relative to the remaining supply reach this condition and revert, even though they would succeed if the discount were computed before the burn.

Code Corrected

`_redeem()` now snapshots `totalSupply()` into a local variable before calling `_burn()` and passes it through to a new `discountPure()` helper. The discount math no longer reads `totalSupply()` live and therefore no longer observes the post-burn supply. The view functions `previewRedeem()`, `_findSharesForAssets()`, and the `discount()` getter were updated to use the same helper with the same supply basis. This resolves all four consequences listed above. The `leftMiddle` subtraction in particular no longer underflows on any redemption path, because it now operates on the pre-burn supply, which is always at least the number of shares being redeemed.

In case the view functions are called with more than `2 * totalSupply()`, there can still be a revert. Such a call has no well-defined answer.

#003 FPS2 Governance Address DoS

Design	High	Version 1	Specification Changed

The FPS2 governance contracts must be deployed at the same address on every chain. The delegation chain that gives the modules their veto power is a set of addresses configured on mainnet: FPS2 delegates to a module, and the modules re-delegate among themselves. BridgedVotes syncs the `delegates` mapping verbatim from mainnet, storing the raw mainnet delegatee addresses, so a module inherits that delegated power on a bridged chain only if it sits at the same address. BridgedVotes also authenticates the cross-chain vote sync purely by address equality:

> Mainnet governance contract is the only valid sender and it should have the same address as this contract (deployed via factory), so we can just check if the sender is address(this)

The modules must also carry the correct mainnet FPS2 address as their helper. Each module passes `defaultHelper()`, which returns its immutable `MAINNET_FPS2`, when exercising FPS2's delegated veto power, and that is what credits FPS2's votes to the module. `MAINNET_FPS2` is set from the `fps2mainnet` argument to `deploy()`.

As deploying through the Arachnid deployer is permissionless, anyone can deploy the InnerFactory and the Governance contracts to their expected addresses. This is OK, because the bytecode enforces that exactly the expected contracts with the expected constructor arguments are deployed. The only exception is the `fps2mainnet` address. It is passed to the deploy function as an argument, so it is not part of the bytecode.

This opens up a DoS scenario: An attacker can deploy the InnerFactory to any chain where it is not yet deployed. Then, the attacker can call `deploy()`, passing an address that is not the real FPS2. This means the contracts cannot execute any governance actions, effectively bricking them. As they are already deployed at the correct addresses, it will be impossible to deploy anything else to those same addresses.

To mitigate this attack, deployment must either be permissioned, or the `fps2mainnet` address must be part of the bytecode (this would require deploying FPS2 at a deterministic address, as the GovernanceFactory must be deployed before FPS2 is).

Specification Changed

The Frankencoin team has decided not to make a change. They state:

> If someone frontruns the FPS2 deployment, we will simply repeat it and treat the first successful deployment as the one we move into. And when deploying Frankencoin to new L2 chains in the future, it should be done in an adjusted version without the baggage of the old versions. And that adjusted version can look at FPS2 votes explicitly without having to rely on the delegations setup in GovernanceFactory. Also note that MainnetVotes.pushFPS2Votes can synchronize votes to any receiving contract on another chain, so we are free to have adjusted BridgedVotes contract on different addresses in the future. The system does not rely on the governance contracts having the same addresses on every L2 chain.

The response clarifies that it is not a requirement for the current version of the contracts to be deployable to additional chains after the initial deployment. Due to this, the DoS scenario described is acceptable. As stated, a griefed initial deploy can be retried using different addresses.

#004 Free Wrap/unwrap Round-Trips Can Delay FPS2 Binding Forever

Design	High	Version 1	Specification Changed

The FPS2 contract accumulates voting power over time. Votes are defined as balance multiplied by holding time. Once FPS2 reaches more than 50% of all FPS1 votes, it is considered binding. However, anyone holding a minority of FPS1 can stop it from reaching binding.

A user can reduce the voting power of FPS2 as follows:

1. **Wrap reduces time without adding votes.** Whenever an address receives additional FPS1, its votes are unchanged, but their tracked holding time is reduced.
2. **Unwrap removes votes proportional to removed balance.** When FPS1 leaves an address, its votes drop in proportion to the balance sent. After a wrap/unwrap cycle, FPS2's balance is restored to its original value, but its holding time is reduced. Votes have been destroyed.

A griefer wrapping and immediately unwrapping an amount equal to 1% of FPS2's total supply removes roughly 1% of the contract's remaining votes per cycle, while getting back all their FPS1. The caller only needs FPS1 to wrap, never spends it, and repeats to drive `relativeVotes(address(this))` arbitrarily low. The only cost is gas.

This has the following consequences:

1. **FPS2 never binds.** Anyone temporarily holding FPS1 can cause FPS2 to never become binding. In this case, the bankrun protection that FPS2 is meant to enforce on the underlying FPS1 never activates.
2. **FPS2 holders cannot redeem.** The same churn resets the FPS2 contract's FPS1 holding duration. FPS1 redemption requires a holding duration of at least 90 days. If the holding duration is below 90 days, FPS2 holders cannot redeem (but they can still unwrap).

Specification Changed

The redemption and unwrapping rules were tightened. Redemption from the FPS2 contract is now allowed only while FPS2 is binding, and `unwrap` is now only allowed if the caller's FPS2 holding duration is greater than the `averageHoldingDuration()` of all users. This restricts unwrapping to long-term holders so that wrap/unwrap round-trips can no longer be done within a short time period.

#005 ERC-4626 Spec Violations in FPS2MintRedeem

Correctness	Medium	Version 1	Code Partially Corrected / Acknowledged

`FPS2MintRedeem` implements ERC-4626, but several functions deviate from the standard's requirements.

1. `convertToAssets()` reflects the redemption discount: `convertToAssets()` applies the redemption discount derived from the recent redemption volume.

The discount is a fee, and it varies with on-chain redemption history. EIP-4626 requires of `convertToAssets()`:

> MUST NOT be inclusive of any fees that are charged against assets in the Vault.
>
> MUST NOT reflect slippage or other on-chain conditions, when performing the actual exchange.

An integrator using `convertToAssets()` as a price oracle sees the per-share value drop under redemption pressure and recover over the recovery period, producing erratic pricing.

2. `totalAssets()` reflects FPS1 price impact: `totalAssets()` prices the contract's entire FPS1 balance through the redemption curve:

```
function totalAssets() public view returns (uint256) {
    return FPS1.calculateProceeds(FPS1.balanceOf(address(this)));
}
```

FPS1 `calculateProceeds()` follows a cubic curve, so the result includes both the 0.3% FPS1 fee and the price impact of liquidating the whole position at once. EIP-4626 defines `totalAssets()` as the

> total amount of the underlying asset that is managed by the Vault

and requires it to be

> inclusive of any fees that are charged against assets in the Vault.

Including the 0.3% fee is correct. However, the price impact is slippage, not a fee. Including it understates the assets managed by the vault. `totalAssets()` should value the FPS1 balance at the marginal redemption price with the 0.3% fee deducted, without the price impact of liquidating the whole position at once.

3. `mint()` may mint more than the requested shares: `mint()` sizes the ZCHF needed for the requested shares, invests through FPS1, and mints `actualShares`, which is at least `shares`. However, `_findAssetsForShares` finds the minimum ZCHF input that yields at least the requested shares, so `actualShares` may slightly exceed shares:

```
uint256 actualShares = FPS1.invest(assets, shares);
_mint(receiver, actualShares);
```

EIP-4626 requires of `mint()`:

> Mints exactly shares Vault shares to receiver by depositing amount of underlying tokens.

To be spec-compliant, the receiver must get exactly the requested shares, even when slightly more FPS1 is minted to FPS2.

4. `withdraw()` may send more than the requested assets: `withdraw()` resolves the minimum share count whose proceeds are at least the requested assets via `_findSharesForAssets()`, then redeems it. Because

shares are discrete, the effective proceeds paid to the receiver can exceed the requested assets. EIP-4626 requires of `withdraw()`:

> Burns shares from owner and sends exactly assets of underlying tokens to receiver.

To be spec-compliant, the receiver must receive exactly the requested `assets`, even when slightly more is received from FPS1 to FPS2.

5. `maxRedeem()` and `maxWithdraw()` ignore FPS1 redemption restrictions: Neither `maxRedeem()` nor `maxWithdraw()` accounts for the fact that redeem reverts when FPS2's average holding period in FPS1 is under 90 days. EIP-4626 requires of `maxRedeem()`:

> MUST factor in both global and user-specific limits, like if redemption is entirely disabled (even temporarily) it MUST return 0.

The equivalent requirement applies to `maxWithdraw()`. Both functions must return zero while `redeem()` and `withdraw()` revert.

6. `convertToShares()` can round up: `convertToShares()` divides the assets by `ask()`, which returns `FPS1.price()`. `FPS1.price()` is itself a floored value:

```
return (VALUATION_FACTOR * zchf.equity() * ONE_DEC18) / totalSupply();
```

Dividing by an already-floored price and flooring again is a double rounding. Because the denominator is rounded down, the quotient is biased up, so the final result can exceed the single-rounded value `floor(assets * totalSupply / (VALUATION_FACTOR * equity))`. EIP-4626 requires of `convertToShares()`:

> MUST round down towards 0.

Relative to the underlying valuation ratio that `FPS1.price()` approximates, `convertToShares()` rounds up rather than down.

Code Partially Corrected

1. `convertToAssets` now uses `FPS1.price()`, which does not include the redemption discount.
2. As of **Version 3**, `totalAssets()` was redefined to `ZCHF.equity() * totalSupply() / FPS1.totalSupply()` and thus refelcts the FPS2 holders' share of the equity pool based on their fraction of the total FPS1 supply. It no longer routes the FPS1 balance through the cubic `calculateProceeds()` curve, so it no longer reflects the FPS1 price impact (slippage) reported above. The 0.3% FPS1 redemption fee is no longer applied.
3. The `mint()` function now mints exactly the requested shares to the user and keeps any excess FPS1 dust that may have been excessively minted in the FPS2 contract.
4. The `withdraw()` function now sends exactly the requested assets to the user and keeps any excess ZCHF dust in the FPS2 contract.
5. `redemptionsEnabled` now requires FPS2 to be binding and have an average holding period of at least 90 days.

Acknowledged

6. `convertToShares` is unchanged and still has the same rounding behavior described above. The Frankencoin team acknowledges and responds:

> Regarding convertToShares(), we argue that FPS1.price() IS the current price of FPS1, and not an approximation. It is true that the price calculation involves flooring the last significant digit. But that does not mean that the returned price is not the true price, it just means that the definition of the current price includes the price being rounded down after 18 digits. The comment on Equity.price() was adjusted accordingly.

#021 FPS2MintRedeem Mishandles Vault-Specific Limits, Violating ERC-4626

Correctness	Medium	Version 2	Code Partially Corrected / Risk Accepted

In FPS2MintRedeem, the preview functions include vault-specific limits that EIP-4626 requires them to ignore, while `maxWithdraw()` ignores a limit it must enforce.

1. `previewRedeem()` returns zero while redemptions are disabled: It returns 0 whenever `redemptionsEnabled()` is false, instead of the proceeds the shares would yield. Disabled redemption is a global limit, and the spec requires of `previewRedeem()`:

> MUST NOT account for redemption limits like those returned from maxRedeem and should always act as though the redemption would be accepted, regardless if the user has enough shares, etc.

In the maxRedeem section of the spec, it is clarified that redemptions being disabled is a limit:

> MUST factor in both global and user-specific limits, like if redemption is entirely disabled (even temporarily) it MUST return 0.

It is also a discrepancy with `previewWithdraw()`, which does not return zero when `redemptionsEnabled()` is false.

2. `previewWithdraw()` reverts above the 10% redemption cap: Through `_findSharesForAssets()`, it caps the search at `totalSupply / 10` and reverts with `RedemptionLimitExceeded` when the requested assets exceed what 10% of supply yields. The cap is a fixed global limit ("like those returned from maxWithdraw", see point 3. below). It is not an "other condition", as it is a fixed chosen limit that sits below the maximum technically feasible value. The spec requires `previewWithdraw()` not to revert due to withdrawal limits:

> MUST NOT revert due to vault specific user/global limits. MAY revert due to other conditions that would also cause withdraw to revert.
>
> MUST NOT account for withdrawal limits like those returned from maxWithdraw and should always act as though the withdrawal would be accepted, regardless if the user has enough shares, etc.

3. `maxWithdraw()` ignores the 10% cap: `maxWithdraw` does not reflect the 10% cap. If a user holds a balance greater than `totalSupply / 10`, it returns a value that would cause `withdraw` to revert. The spec requires of `maxWithdraw()`:

> MUST return the maximum amount of assets that could be transferred from owner through withdraw and not cause a revert, which MUST NOT be higher than the actual maximum that would be accepted (it should underestimate if necessary).

This is the mirror image of issues 1 and 2: the 10% cap is a global limit that `previewWithdraw()` wrongly accounts for, yet `maxWithdraw()`, which must account for it, does not.

Code Partially Corrected

1. `previewRedeem()` no longer returns zero while redemptions are disabled, but returns the proceeds the shares would yield.

3. `maxWithdraw()` now enforces the 10% cap: it returns `previewRedeem()` of `min(totalSupply() / 10, balanceOf(owner))`, or 0 when `redemptionsEnabled()` is false. It therefore no longer overstates the withdrawable amount for owners holding more than 10% of supply.

Risk Accepted

2. `previewWithdraw()` is unchanged and still reverts with `RedemptionLimitExceeded` via `_findSharesForAssets()` when the requested assets exceed the proceeds of 10% of supply.

The Frankencoin team has decided to leave this behavior unchanged and accepts it as a conscious minor deviation from the ERC-4626 specification: `previewWithdraw()` reverts at the same `totalSupply() / 10` bound as `withdraw()` itself, and the now-corrected `maxWithdraw()` acts as the ceiling for integrators.

#022 Inverted Holding-Duration Check in Unwrap

Correctness	Medium	Version 2	Code Corrected

`unwrap()` in FPS2 incorrectly reverts when the caller's FPS2 holding duration is **at least** the average:

```
if (holdingDuration(msg.sender) >= averageHoldingDuration()) revert NotFIFO();
```

It should revert when the holding duration is **below** the average.

The check is meant to restrict unwrapping to long-term holders, preventing the free wrap/unwrap round-trip issue. Instead, it blocks holders who have held longer than average and continues to allow immediate wrap/unwrap.

Code Corrected

`unwrap()` now correctly checks that the holding duration is at least the average.

#023 maxRedeem Returns an Amount Instead of a Share Count

Correctness	Medium	Version 2	Code Corrected

In **Version 2**, FPS2MintRedeem's `maxRedeem()` was changed to return `previewRedeem(balanceOf(owner))` instead of `balanceOf(owner)` (as in **Version 1**). previewRedeem returns an asset amount, but it should be a share count.

Code Corrected

The function has been changed to use `balanceOf(owner)` again.

#006 Arbitrarily Old Vote Snapshots Can Be Applied

Design	Low	Version 1	Risk Accepted

BridgedVotes is the source of truth of voting power on bridged chains. Its `votes()` and `totalVotes()` feed `checkQualified()`, which grants veto power when a holder's delegated votes exceed 1% of `totalVotes()`. Both values are set entirely from messages pushed over CCIP by MainnetVotes through `pushFPS2Votes()`.

Root cause: `_ccipReceive()` in BridgedVotes applies every accepted message unconditionally. It overwrites `_fps2TotalVotes` and each `_fps2Votes` entry with the payload, carrying no notion of recency. It includes no sequence number or timestamp check.

Messages do not expire: A committed CCIP message stays executable forever after it is sent. If its initial execution fails, the message becomes available for manual execution. `pushFPS2Votes()` forwards caller-supplied `extraArgs`, so a deliberately low `gasLimit` can make the receiver call run out of gas, leaving the message pending for manual execution. Anyone can call `pushFPS2Votes()` and pay the fee, so an attacker can deliberately create many messages frozen at a snapshot of their choosing and execute them at a moment of their choosing. Anyone else can also execute them to prevent this.

Consequences: Applying an old message reverts the bridged state to a stale snapshot. This breaks the invariant that `votes()` and `totalVotes()` reflect current mainnet FPS2 holdings. This presents the following problems:

1. **Forge veto qualification:** A holder who has since reduced their FPS2 position holds a message built from the larger old balance. Executing it restores the inflated `_fps2Votes` entry, after which `checkQualified()` passes and the holder vetoes with power they no longer have.

2. **Censor honest vetoers:** If `totalVotes()` has since fallen, a message carrying the larger old total raises the `QUORUM * totalVotes()` bar. Replaying it pushes legitimate holders below the 1% threshold so their `checkQualified()` reverts. The legitimate holder must sync votes again to correct this.

3. **Backrun a corrective resync:** An honest party who resyncs to repair the state can be backrun with another stale message again, undoing the correction as long as a stale executable message exists.

Honest users can monitor for stale messages and execute them manually to prevent the above consequences.

The same problems are also present in the deployed BridgedGovernance contract, which implements the same functionality for FPS1.

Risk Accepted

No code change was made. `_ccipReceive` in BridgedVotes still applies each accepted message unconditionally, so an arbitrarily old stuck message remains executable at a time of the caller's choosing. Frankencoin intends to observe the chain for failed sync messages and execute them once detected, ensuring that messages do not stay in a stuck state for a long time. They state:

> The risk is accepted under the assumption that it is more expensive for an attacker to initiate a stuck sync on mainnet than it is to manually executed the sync on the L2 chain in order to flush out the intentionally stuck messages. The defense against such an attack would therefore be to watch the blockchain for stuck sync messages and flush them once detected.

#007 Binary Search Assumes Monotonicity of Effective Proceeds

Correctness	Low	Version 1	Code Corrected

`_findSharesForAssets()` in FPS2MintRedeem binary-searches for the minimum FPS2 shares whose effective proceeds reach the requested `assets`. Its objective `calculateEffectiveProceeds()` multiplies FPS1 `calculateProceeds()`, which increases in the share count, with `discount()`, a 4th-power factor that decreases in the share count and reaches zero at `2 * totalSupply()`. The product rises to a peak and then falls back to zero, so the objective is not monotonic. The expand loop and the binary search both assume it increases monotonically.

```
uint256 hi = lo + lo / 5 + 1;
while (calculateEffectiveProceeds(recent, hi, FPS1.calculateProceeds(hi)) < assets) {
```

```
    hi *= 2;
}
```

This has the following consequences:

1. **A feasible withdrawal can revert:** When the given `assets` value sits just below the peak, the satisfying share counts span less than one doubling step. The `hi *= 2` loop steps over them, `hi` grows past `2 * totalSupply()`, and the next `discount()` evaluation underflows. Even though a solution exists, the function reverts.

2. **An unachievable request reverts opaquely:** When `assets` exceeds the peak, no share count satisfies it. The loop doubles `hi` past `2 * totalSupply()` and reverts with a Panic or the FPS1 `calculateProceeds()` `too many shares` requirement. It could instead fail with a descriptive error.

3. **Binary search may not return smallest solution:** Because the objective is non-monotonic, there can be two solutions (one before the peak and one after). The binary search may return either one, although only the smaller one is the intended return value.

Note that the `calculateEffectiveProceeds()` function is monotonic for small values and only reaches its peak once a redemption attempts to redeem a large percentage of all FPS2 at once. This limits the practical impact, as users are incentivized to split large redemptions to avoid paying large discount fees.

> **Code Corrected**
>
> The `_findSharesForAssets()` function was adjusted to revert in case the requested value requires redeeming more than 10% of the `totalSupply` of FPS2. This ensures that the range the search is performed in is before the peak.

#008 GovernanceFactory Does Not Implement Correct Interface, Blocking FPS2 Deployment

Correctness	Low	Version 1	Code Corrected

The FPS2 constructor decodes two return values from the factory:

```
(address votes, address helper) = factory.deploy(address(this));
```

However, `deploy()` in GovernanceFactory and InnerFactory only returns a single address. `deploy()` must return both `votes` and `helper`.

> **Code Corrected**
>
> The FPS2 constructor has been adjusted to now decode only a single return value from `deploy()`, matching the factory.

#009 MutualDestruction Event Reports Cumulative Votes Instead of the per-Target Amount

Correctness	Low	Version 1	Code Corrected

In AccumulatingVotesToken, the `attack()` function's loop accumulates the running total of destroyed votes into `destroyedVotes` and emits one event per target:

```
for (uint256 i = 0; i < targets.length && destroyedVotes < budget; i++) {
    destroyedVotes += _reduceVotes(targets[i], budget - destroyedVotes);
    emit MutualDestruction(msg.sender, targets[i], destroyedVotes);
}
```

The event declares its third parameter as the votes destroyed for the given target:

```
event MutualDestruction(address indexed initiator, address indexed target, uint256
    votesDestroyed);
```

However, the emitted value is the cumulative sum over all targets processed so far, not the amount returned by `_reduceVotes()` for `targets[i]`. For every target after the first, `votesDestroyed` overstates the votes actually destroyed for that target.

> **Code Corrected**
>
> The code has been updated to emit the correct amount.

#010 Redemption Discount Recovers Slower than Intended

Design	Low	Version 1	Specification Changed

In FPS2MintRedeem, the redemption discount is driven by `weightedRecentRedemptions()`, which is meant to decay the recently redeemed volume linearly to zero over `RECOVERY_PERIOD` (7 days). The contract documentation states: "This spread is closed linearly over the course of a week."

Root cause: `weightedRecentRedemptions()` decays `recentlyRedeemed` at the constant rate `recentlyRedeemed / RECOVERY_PERIOD`:

```
return recentlyRedeemed * (RECOVERY_PERIOD - elapsed) / RECOVERY_PERIOD;
```

Every redemption then rebases `recentlyRedeemed` to the current decayed value and resets `redemptionAnchor` to the present, in `_redeem()`:

```
uint256 recent = weightedRecentRedemptions();
...
recentlyRedeemed = uint192(recent + shares);
redemptionAnchor = uint64(block.timestamp);
```

Because the decay rate is proportional to `recentlyRedeemed`, each rebase lowers the absolute decay rate. Only the remaining weight continues to decay, and the original 7-day schedule is never honored.

Worked example: a single `redeem(target, 0)` at 3.5 days finds `recent` at 50% of the original volume, rebases `recentlyRedeemed` to that 50%, and resets the time. At 7 days after the original redemption the weight is `0.5 * 0.5 = 25%` instead of 0. Repeating the call continuously converts the intended linear decay into exponential decay, leaving `e^{-1} ≈ 36.8%` of the original volume after a full week.

As a result, the redemption discount decay is slower than intended and depends on how often redemptions occur.

> **Specification Changed**
>
> Frankencoin decided to keep the decay calculation as is, accepting that the schedule can become an exponential decay instead of linear decay. Full recovery takes longer in periods of frequent redemptions.

However, the initial recovery after a large redemption event is similarly quick for both linear and exponential decay.

#011 totalAssets Counts Unredeemable FPS1

Correctness	Low	Version 1	Specification Changed

`totalAssets()` in FPS2MintRedeem prices the contract's entire FPS1 balance.

The ERC-4626 spec defines totalAssets as:

> Total amount of the underlying asset that is "managed" by Vault.

It is possible for the FPS1 balance of FPS2 to be larger than the FPS2 `totalSupply`, for example if there was a direct transfer of FPS1 tokens. These extra FPS1 can never be redeemed, and should not be counted in totalAssets, as their underlying Frankencoins are not "managed" by the vault.

Specification Changed

As of **Version 3**, the `totalAssets()` function no longer uses the contract's FPS1 balance. It has been changed to the following:

```
function totalAssets() public view returns (uint256) {
        return ZCHF.equity() * totalSupply() / FPS1.totalSupply();}
```

#012 Very Large Redemptions Can Reach Negative Marginal Price

Design	Low	Version 1	Risk Accepted

In FPS2MintRedeem, `_redeem()` pays out `effectiveProceeds = rawProceeds * discount(recent, shares)`. The discount factor uses a 4th-power curve in `discount()`, evaluated at the midpoint `currentSupply - shares / 2`. As `shares` grows, `rawProceeds` from `FPS1.redeem()` grows roughly in proportion, but the discount factor decays with the fourth power of the redeemed fraction. The decay eventually dominates, so the effective proceeds are non-monotonic in `shares`.

Negative marginal price: approximating `rawProceeds` as proportional to `shares` and taking `recent` as zero, the payout is proportional to `x * (1 - x/2)^4` where `x` is the redeemed fraction of total supply. Depending on how much of the FPS1 supply is held by FPS2, this peaks between `x = 0.3` and `x = 0.4` and decreases thereafter. Redeeming 60% of supply in one call yields about 0.144 units against 0.164 at 40%. Past the peak the marginal redemption price is negative: each additional share burned reduces the total ZCHF returned.

A user redeeming more than roughly 30% of the supply in a single call therefore receives strictly less than they would by redeeming a smaller amount, and burns more shares for it.

Risk Accepted

Frankencoin accepts the non-monotonic payout. The frontend will warn users that splitting a large redemption into smaller amounts yields more ZCHF, leaving the redemption size to the user's discretion.

The 10% cap in `_findSharesForAssets()` keeps the asset-based `withdraw()` path on the increasing part of the curve, reverting with `RedemptionLimitExceeded` above 10% of supply. The share-based `redeem()` entry points are intentionally uncapped, so redemptions past the peak remain reachable and are addressed by the frontend warning rather than a hard cap.

#024 `totalAssets()` Reverts Once FPS2 Owns Nearly the Entire FPS1 Supply

Correctness	Low	Version 1	Specification Changed

In FPS2MintRedeem, `totalAssets()` values the contract's FPS1 holding by routing the full balance through the FPS1 `calculateProceeds` function:

```
function totalAssets() public view returns (uint256) {
    return FPS1.calculateProceeds(FPS1.balanceOf(address(this)));
}
```

In Equity, `calculateProceeds()` has an edge case where it rejects inputs close to the total share supply:

```
uint256 totalShares = totalSupply();
require(shares + ONE_DEC18 < totalShares, "too many shares");
```

The call reverts as soon as the passed amount reaches `totalShares - ONE_DEC18`. Since FPS2 is itself an FPS1 holder, `FPS1.balanceOf(address(this))` grows toward `FPS1.totalSupply()` as FPS1 holders wrap into FPS2. Once fewer than one whole FPS1 token (`1e18`) is held outside FPS2, `totalAssets()` reverts. As FPS2 is intended to fully replace FPS1 eventually, this condition may be reached eventually.

EIP-4626 requires `totalAssets()` to not revert:

> MUST NOT revert.

A reverting `totalAssets()` breaks any integrator or aggregator that reads it.

Specification Changed

The definition of `totalAssets()` has been changed. It no longer uses `calculateProceeds()`. Instead, it is calculated directly from Frankencoin's equity value:

```
function totalAssets() public view returns (uint256) {
    return ZCHF.equity() * totalSupply() / FPS1.totalSupply();}
```

#028 `denyUnannouncedMinter` Can Be Disabled

Correctness	Low	Version 1	Code Partially Corrected / Risk Accepted

In MinterGovernance, `suggestMinter()` sets `announcements[_minter] = block.timestamp` and never clears it. If the minter is then vetoed on the Frankencoin level, it can be proposed again with the shorter minimum application period of Frankencoin. `denyUnannouncedMinter()` in MinterGovernance cannot be called on the minter anymore, as it is still present in `announcements`.

```
if (announcements[minter] != 0) revert MinterCorrectlyAnnounced();
```

Attack scenario:

1. The attacker announces minter `M` through `suggestMinter()`, setting `announcements[M]`.

2. `M` is vetoed via `denyMinter()`. `minters[M]` in Frankencoin is deleted, `announcements[M]` in MinterGovernance is not.
3. The attacker calls `ZCHF.suggestMinter(M, ...)` directly, using Frankencoin's shorter base `MIN_APPLICATION_PERIOD` (14 days) instead of the 60 days MinterGovernance enforces.
4. `denyUnannouncedMinter(M)` reverts and `checkReward(M)` returns 0, so no bot is incentivized to veto `M`.

The cost for the attacker is an additional 1200 ZCHF (cost of announcing the minter).

For any address ever announced and subsequently removed, the 60-day minimum period can be bypassed while avoiding the incentivized permissionless enforcement. Qualified FPS/FPS2 holders can still veto via `denyMinter()`. This shifts part of the monitoring burden that was supposed to be alleviated through `denyUnannouncedMinter` back to FPS/FPS2 holders.

Code Partially Corrected

The `denyMinter` function in MinterGovernance now deletes the vetoed minter from the announcements.

Risk Accepted

`denyMinter()` can also be called directly on the ZCHF contract by a qualified FPS1 holder, which does not delete the mapping in MinterGovernance. This can still be used to circumvent the incentivized `denyUnannouncedMinter()`, meaning FPS2 holders still need to monitor for unannounced minters and cannot fully rely on incentivized bots.

However, the remaining problem only happens if a veto is made directly by a qualified FPS1 holder. This is no longer expected once FPS2 becomes binding, as shoot can be used to remove all FPS1 votes that are held by any address other than FPS2.

#013 `shoot()` Does Not Explicitly Reject FPS2 as Target

Informational	Version 1	Code Corrected

In FPS2, `shoot()` uses `FPS1.kamikaze()` on a `target`. There is no explicit check that the `target` is not FPS2.

Due to a sanity check in FPS1, a self-shoot is inadvertently caught and will revert. Adding an explicit check would be more future-proof and could surface a more informative error message.

```
require(destroyedVotes > 0); // sanity check
```

Code Corrected

An explicit revert has been added.

This issue was initially raised by the Frankencoin team during a joint code walkthrough.

#014 Abstract Leadrate Does Not Inherit ILeadrate

Informational	Version 1	Code Corrected

The AbstractLeadrate correctly implements the ILeadrate interface but does not inherit from it. Inheriting it would allow the compiler to throw an error if the interface is accidentally broken in a future version.

Code Corrected

AbstractLeadrate now inherits ILeadrate.

#015 Suggesting a Minter Reverts Despite Paying the Contract's Stated Minimum Fee

Informational | **Version 1** | **Specification Changed**

In MinterGovernance, `suggestMinter()` checks that `_applicationFee >= MIN_APPLICATION_FEE` (1000 ZCHF), then forwards only `_applicationFee - rewardPoolRefill` to Frankencoin. Frankencoin's `suggestMinter()` independently requires `_applicationFee >= MIN_FEE`, also 1000 ZCHF, and reverts with `FeeTooLow` otherwise.

The actual minimum that succeeds is `MIN_APPLICATION_FEE + rewardPoolRefill`.

The `MIN_APPLICATION_FEE` check is currently misleading. It could be updated to reflect the actual amount needed for the call to succeed.

Specification Changed

The `MIN_APPLICATION_FEE` has been increased to 1200, which now makes it a meaningful check. The reward pool refill logic has also been changed to always require a fee of 1200, independent of the state of the reward pool.

#016 Inconsistent Function Naming

Informational | **Version 1** | **Code Corrected**

In FPS2MintRedeem, the `deposit()` and `redeemExpected()` functions are alternative versions of `deposit()` and `redeem()`. They take an `expectedShares` / `expectedProceeds` argument that provides slippage protection. While the two functions behave similarly, the naming is inconsistent. `deposit()` is an overload that keeps the same name while changing the parameters, whereas `redeemExpected()` adds an "expected" suffix instead.

Applying the same naming convention to both functions could help avoid confusion.

Code Corrected

The slippage-protected deposit overload was renamed to `depositExpected`, matching the existing `redeemExpected`. Moreover, the function now also takes the `recipient` address as a function parameter.

#017 Non-Indexed Event Parameters

Informational | Version 1 | Code Corrected

The following events omit indexed parameters (non-exhaustive list): Indexing relevant fields makes them searchable in the logs, which can be useful for off-chain applications that need to track specific events or filter them based on certain criteria.

- **FPS2.sol**
  1. `Wrapped`: address `who`
  2. `Unwrapped`: address `who`
  3. `Shot`: address `target`
- **BridgedVotes.sol**
  1. `FPS2VotesReceived`: bytes32 `messageId`

Adding indexing to some of these events may be helpful for off-chain integrators.

> **Code Corrected**
> All mentioned events are now indexed.

#018 Outdated and Incorrect Documentation Comments

Informational | Version 1 | Code Corrected

Several comments no longer match the code:

1. **Veto threshold in Governance.** The contract header states "Veto power is reached with 2% of the votes." The active threshold is QUORUM, which has been changed to 1%. The comment is outdated.
2. **Application period in MinterGovernance.** MIN_APPLICATION_PERIOD is 60 days, while suggestMinter documents that it wraps Frankencoin "to enforce a 90 days application period." The constant and the comment disagree.
3. **Candidate selection in MinterGovernance `checkReward()`.** The `@dev` comment instructs callers to look for MinterApplied events "whose minter parameter is not this contract's address." This is incorrect. A minter announced through this contract will not have this contract as the minter parameter, yet it should still be excluded. The correct filter is to drop minters that have a corresponding MinterAnnounced event.
4. **Candidate selection in MinterGovernance `denyUnannouncedMinter()`.** The `@dev` comment is the same incorrect text as in `checkReward()`.
5. **NatSpec of `denyUnannouncedMinter()`.** The `@notice` says "after MinterGovernance was deployed." The actual prerequisite is that FPS2 holds at least 2% of FPS1 voting power.
6. **ticksAnchor in AbstractLeadrate**. The comment says "in bips * seconds, `uint88` allows up to". The comment is truncated. Also the `ticksAnchor` is `uint64`, not `uint88`.
7. **`_findAssetsForShares` comment (Version 2).** Documents a binary search, but the function now computes the result in closed form.

> **Code Corrected**
> All items were addressed accordingly.

#019 Unused Error

Informational	Version 1	Code Corrected

The FPS2 contract defines the `CannotWipeSelf` error, but it is never used in the codebase.

> **Code Corrected**
> The error has been removed.

#020 Unused Imports

Informational	Version 1	Code Corrected

The following imports are unused in their respective files:

1. In `FPS2.sol`: `IEquity` is imported but not used.
2. In `CCIPGovernance.sol`: `IFrankencoin` and `IPosition` are imported but not used.

> **Code Corrected**
> The unused imports were removed from the codebase.

#025 CCIP Rate Limit Increases Could Be Delayed

Informational	Version 1	Acknowledged

`CCIPGovernance.applyRateLimit` calls into the `CCIP_ADMIN` contract. Setting the rate limit can be done instantly by anyone with enough voting power to pass the quorum check.

There could be benefits to making instant setting only possible for decreasing the rate limit (or a simpler implementation could allow instant setting only for setting it to 0), and applying a delay for increasing. This would allow instant reaction to an emergency, which would then be harder to revert by an attacker. It is a tradeoff, as it would open up a new griefing possibility. A malicious user could set the limit to zero, which would take time to undo.

Note that within FPS2 there may be cases where very little voting power is required to access this instant setter. When FPS2 has 2% of voting power within FPS1, the FPS2 quorum can be reached with 1% of FPS2 votes, which can then be around 0.02% of all FPS1 votes.

> **Acknowledged**
> The Frankencoin team acknowledged the concern and has decided to address it in a future version, leaving the current code unchanged.
>
> They state:

> Yes, this absolutely makes sense. However, since we have already reached the byte size limit for the GovernanceFactory, we decided to implement that change independently through a general update of the CCIPAdmin module. This update would make both the old deployed CCIPAdmin modules obsolete as well as the new CCIPGovernance.

#026 previewMint of Zero Shares Returns a Positive ZCHF Amount

Informational **Version 2** **Acknowledged**

In FPS2MintRedeem, `previewMint()` forwards to `_findAssetsForShares()`, which always adds a `worstCaseUndershoot` safety margin to the requested share count before computing the required ZCHF:

```
uint256 worstCaseUndershoot = 3 * fps1Supply / 1e18;
uint256 growthFactor = _divD18(fps1Supply + shares + worstCaseUndershoot, fps1Supply);
```

When `shares` is 0 the margin is still applied, so `growthFactor` exceeds 1e18 and the returned `capitalNeeded` is strictly positive. As a result `previewMint(0)` reports a non-zero asset cost for minting zero shares.

> **Acknowledged**
>
> Frankencoin decided not to make a change, stating:
>
> > No change needed. This function always returns a marginal amount more than actually needed in order to always conform to the 4626 standard. Returning 0.0001 when the optimial result is 0.0000 is equally wrong as returning 7.0001 when the optimal result is 7.0000.

#027 Unused Variable in MainnetVotes

Informational **Version 2** **Code Corrected**

In MainnetVotes, the `FPS1` immutable is declared but never assigned in the constructor and never read.

> **Code Corrected**
>
> The immutable has been removed.

8 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.