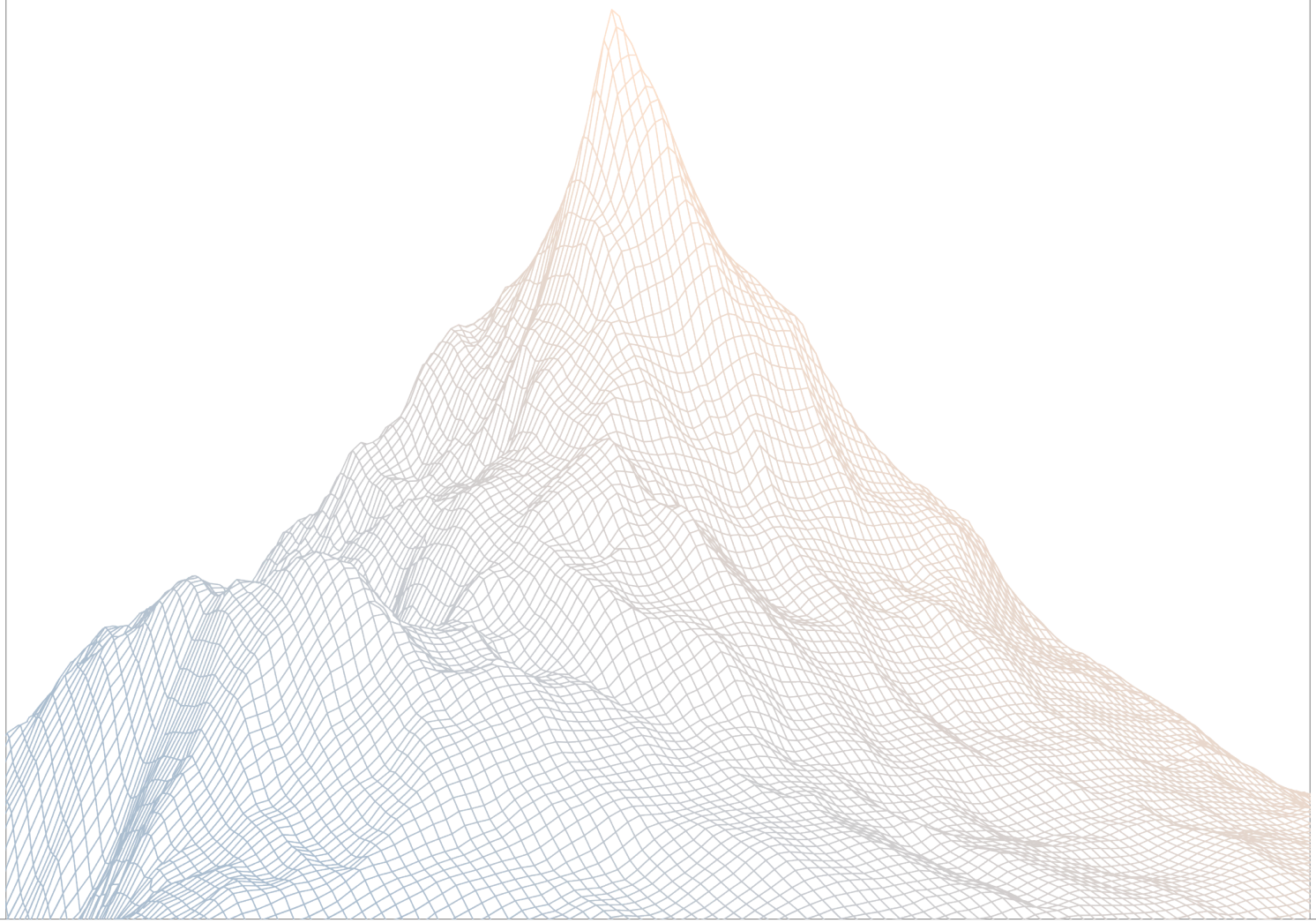
Zenith

Fiamma

Smart Contract Security Assessment

VERSION 1.1



AUDIT DATES: July 15th to July 22th, 2025

AUDITED BY: EV_om
zzykxx

Contents

1 Introduction

1.1 About Zenith

Zenith assembles auditors with proven track records: finding critical vulnerabilities in public audit competitions.

Our audits are carried out by a curated team of the industry's top-performing security researchers, selected for your specific codebase, security needs, and budget.

Learn more about us at https://zenith.security.

1.2 Disclaimer

This report reflects an analysis conducted within a defined scope and time frame, based on provided materials and documentation. It does not encompass all possible vulnerabilities and should not be considered exhaustive.

The review and accompanying report are presented on an "as-is" and "as-available" basis, without any express or implied warranties.

Furthermore, this report neither endorses any specific project or team nor assures the complete security of the project.

1.3 Risk Classification

SEVERITY LEVEL	IMPACT: HIGH	IMPACT: MEDIUM	IMPACT: LOW
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

2

Executive Summary

2.1 About Fiamma

We are a team of cryptographers, blockchain developers, and fintech professionals pioneering BitVM2—the Bitcoin Virtual Machine—ushering in a new era of trust-minimized Bitcoin infrastructure and cross-ecosystem bridges built with Bitcoin finality for the first time.

We were the first to enable Bitcoin to verify ZKPs optimistically under BitVM2. This milestone lays the groundwork for the most trust-minimized foundation to bring BTC into broader DeFi, Web3, and real-world applications. We hold a deep conviction: Zero-knowledge technology is the key to unlocking Bitcoin's full potential.

After two rounds of testnet launches for our BitVM-powered bridge, it became clear that another frontier must be tackled: user experience—especially when striving for both self-custody and trust minimization.

That's why we will also launch Fiamma One soon, a non-custodial superapp delivering one-click BTC earning—a seamless, secure first step toward unlocking broader BTC use cases like non-custodial lending, stablecoin and RWA strategies, and more.

This is only the beginning. We're building the rails for the next chapter of finance—rooted in Bitcoin, connected to all.

2.2 Scope

The engagement involved a review of the following targets:

Target	starknet-contracts
Repository	https://github.com/fiamma-chain/starknet-contracts
Commit Hash	`2371f33fe8a7de733f137c75de91d4021570fff7`
Files	`packages/*`

2.3 Audit Timeline

July 15, 2025	Audit start
July 22, 2025	Audit end
July 25, 2025	Report published

2.4 Issues Found

SEVERITY	COUNT
Critical Risk	0
High Risk	1
Medium Risk	2
Low Risk	2
Informational	5
Total Issues	**10**

3

Findings Summary

ID	Description	Status
H-1	Old committee members retain signing privileges after committee updates	Resolved
M-1	No way to modify fee rate or operator ID for pending withdrawals	Acknowledged
M-2	submit_block_headers() compares the work of single blocks instead of total work when crossing difficulty boundaries	Resolved
L-1	Incorrect difficulty adjustment boundary check deviates from Bitcoin consensus rules	Resolved
L-2	btc_utils::bits_to_target() rejects valid block headers whose exponents are 31 or 32	Resolved
I-1	Inaccurate Reorg event parameter	Resolved
I-2	Redundant threshold check after early return logic	Resolved
I-3	Documentation inconsistency regarding owner permissions in burn function	Resolved
I-4	Outdated OpenZeppelin dependencies	Resolved
I-5	bitvm_bridge::mint() edge-case allows double-minting	Resolved

4

Findings

4.1 High Risk

A total of 1 high risk findings were identified.

[H-1] Old committee members retain signing privileges after committee updates

SEVERITY: High	IMPACT: High
STATUS: Resolved	LIKELIHOOD: High

Target

- committee_verifier.cairo

Description:

The `SimpleCommitteeVerifier` contract manages a committee of signers who can authorise bridge operations by providing threshold signatures. The contract allows the owner to update the committee membership through the `set_committee()` function.

The implementation uses a storage mapping `committee_members` that maps public keys to boolean values, indicating whether a given public key is an authorised committee member. When verifying signatures in `verify_dest_script_hash()`, the contract checks if each signer is a valid committee member by reading from this mapping.

Interestingly, the code contains a comment suggesting that "in production, you might want to track and clear old members" — advice that appears to have been overlooked in the current implementation. When setting new committee members, the function only writes `true` for the new members but fails to set `false` for the previous committee members who are being removed:

```
// Add new members
for i in 0..members.len() {
    let member = *members.at(i);
    assert(!member.is_zero(), Errors::ZERO_PUBLIC_KEY);
    self.committee_members.write(member, true);
}
```

This means that if the committee is updated from members {A, B, C, D} to {X, Y, Z}, the storage will contain {A: true, B: true, C: true, D: true, X: true, Y: true, Z: true}, effectively

granting signing privileges to both the old and new committee members.

The highest impact scenario occurs when a committee member is removed due to being compromised or no longer trusted. Despite the owner's intention to revoke their signing privileges by updating the committee, the removed member retains the permissions.

Additional consequences include:

- The actual signing threshold becomes meaningless as the effective committee size grows with each update
- Operational confusion as the actual committee composition diverges from the intended one
- Inability to rotate keys or remove compromised signers, breaking a fundamental security requirement

Recommendations:

Replace the mapping-based approach with an array to store committee members. This naturally solves the cleanup issue as the entire array is replaced when updating the committee:

```
#[storage]
struct Storage {
    #[substorage(v0)]
    ownable: OwnableComponent::Storage,
    /// Mapping of committee member public keys to their status
    committee_members: Map<u256, bool>,
    /// Array of committee member public keys
    committee_members: Array<u256>,
    /// Minimum number of signatures required
    threshold: u32,
}

fn set_committee(ref self: ContractState, members: Array<u256>, threshold:
    u32) {
    // Only owner can set committee
    self.ownable.assert_only_owner();

    // Validate inputs
    assert(members.len() > 0, Errors::EMPTY_COMMITTEE);
    assert(threshold > 0 && threshold <= members.len(),
    Errors::INVALID_THRESHOLD);

    // Clear existing committee members (simple approach: just overwrite)
    // In production, you might want to track and clear old members
```

```
    // Add new members
    for i in 0..members.len() {
        let member = *members.at(i);
        assert(!member.is_zero(), Errors::ZERO_PUBLIC_KEY);
        self.committee_members.write(member, true);
    }
    // Validate new members
    for i in 0..members.len() {
        let member = *members.at(i);
        assert(!member.is_zero(), Errors::ZERO_PUBLIC_KEY);
    }

    // Replace entire committee (automatically clears old members)
    self.committee_members.write(members.clone());

     // Set threshold
     self.threshold.write(threshold);

     // Emit event
     self.emit(CommitteeSet { members, threshold });
 }
```

Update the verification logic to check array membership:

```diff
 // In verify_dest_script_hash()
    // Skip non-committee members
    if !self.committee_members.read(signer) {
        continue;
    }
    // Skip non-committee members
    if !self._is_committee_member(signer) {
        continue;
    }

// Add helper function
fn _is_committee_member(self: @ContractState, signer: u256) -> bool {
    let members = self.committee_members.read();
    for i in 0..members.len() {
```

```
        if *members.at(i) == signer {
            return true;
        }
    }
    false
}
```

If needed, implement analogous `add_committee()` and `remove_committee()` functions.

Fiamma: Resolved with PR-1

Zenith: Verified, members have to be explicitly removed.

4.2 Medium Risk

A total of 2 medium risk findings were identified.

[M-1] No way to modify fee rate or operator ID for pending withdrawals

SEVERITY: Medium	IMPACT: Medium
STATUS: Acknowledged	LIKELIHOOD: Medium

Target

- bitvm_bridge.cairo

Description:

The BitVM Bridge facilitates bridging between Bitcoin and Starknet by allowing users to mint tokens through Bitcoin deposits and burn tokens to initiate Bitcoin withdrawals. The withdrawal process is handled through the `burn()` function, which accepts parameters including the destination Bitcoin address, fee rate, withdrawal amount, and operator ID.

When a user initiates a withdrawal by calling `burn()`, they must specify a `fee_rate` (satoshis per byte) and an `operator_id` for the Bitcoin transaction. The function burns the specified amount of tokens and emits a `Burn` event that bridge operators monitor to process the actual Bitcoin transaction off-chain.

When a withdrawal is initiated, there is no mechanism to modify the `fee_rate` or `operator_id`. This creates several problematic scenarios:

1. **Insufficient fee rate**: If a user sets a `fee_rate` that becomes insufficient due to Bitcoin network congestion, the withdrawal transaction may remain unconfirmed indefinitely. Bitcoin fee markets are volatile, and a fee rate that seems reasonable at burn time may become inadequate hours or days later.

2. **Unresponsive operator**: If the specified `operator_id` corresponds to an operator that becomes unavailable, malicious, or refuses to process the transaction, the withdrawal cannot be reassigned to another operator.

Users have no way to update these parameters after burning their tokens, meaning their funds could remain locked in the withdrawal queue indefinitely.

Recommendations:

Implement a mechanism to allow users to update withdrawal parameters after submission. Consider one or more of the following approaches:

1. **Withdrawal request system**: Instead of immediately burning tokens, create a two-step process where users first create a withdrawal request with on-chain confirmation from the operator.

2. **Fee bump mechanism**: Allow users to increase the fee rate of pending withdrawals by burning additional tokens to cover the difference.

3. **Operator reassignment**: Implement a timeout mechanism where withdrawals can be reassigned to different operators or picked up by any other if not processed within a reasonable timeframe.

Fiamma: Acknowledged. This suggestion is good, but currently, we haven't implemented it yet. We will adapt the corresponding bridge backend code and frontend code next time. Right now, our frontend code calls the contract by combining real-time recommended rates with user-selected rates.

[M-2] `submit_block_headers()` compares the work of single blocks instead of total work when crossing difficulty boundaries

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Low

Target

- btc_mirror

Description:

The function btc_mirror::submit_block_headers() requires the work of the new chain to be greater than the work of the re-orged chain when crossing difficulty boundaries.

The work is calculated via btc_mirror:: _get_work_in_period(), which internally executes btc_utils::compute_work_from_target():

```
pub fn compute_work_from_target(target: u256) -> u256 {
    (~target / (target + 1_u256)) + 1_u256
}
```

This computes the work of a single block in the new period and the work of a single block in the previous period but it should calculate the **total** work of the previous chain and the **total** work of the new chain.

A consequence of this is the light client rejecting block headers crossing difficulty boundaries where the target difficulty decreases.

One example of difficulty decrease is block 901152.

To test this you can apply the following patch which attempts to submit block headers `901151`, `901152`, `901153`:

```
diff --git a/packages/btc_light_client/tests/test_btc_mirror.cairo
    b/packages/btc_light_client/tests/test_btc_mirror.cairo
index 0118595..286407b 100644
-- a/packages/btc_light_client/tests/test_btc_mirror.cairo
++ b/packages/btc_light_client/tests/test_btc_mirror.cairo
@@ -4,10 +4,10 @@ use btc_light_client::btc_mirror::{IBtcMirrorDispatcher,
```

```
    IBtcMirrorDispatcherTra
 use snforge_std::{ContractClassTrait, DeclareResultTrait, declare};
 use crate::utils::from_hex;
 //start at block #717694, two blocks before retarget
const INITIAL_BLOCK_HEIGHT: u32 = 717694_u32;
const INITIAL_BLOCK_HASH: u256 =
 0xedae5e1bd8a0e007e529fe33d099ebb7a82a06d6d63d0b0000000000000000;
const INITIAL_BLOCK_HEIGHT: u32 = 901150_u32;
const INITIAL_BLOCK_HASH: u256 =
 0xc7cf24cef41ca7411a8406164c1c2bd17c8cac2b000d020000000000000000;
 const INITIAL_BLOCK_TIME: u32 = 1641627092_u32;
const INITIAL_TARGET: u256 =
 0x00000000000000000B98AB0000000000000000000000000000000000000000;
const INITIAL_TARGET: u256 =
 0x0000000000000000000000000000000000000000237740000000000000000;

 fn deploy_btc_mirror() -> IBtcMirrorDispatcher {
     let btc_mirror_class = declare("BtcMirror").unwrap().contract_class();
@@ -161,6 +161,23 @@ fn test_submit_block_headers_success() {
     assert_eq!(target, expected_target);
 }

#[test]
fn test_submit_block_headers_custom() {
    let btc_mirror = deploy_btc_mirror();
    let mut block_header: Array<u8> = array![];

    let mut height = INITIAL_BLOCK_HEIGHT + 1;
    let block_header_custom0 =
create_block_custom
("0000b927c7cf24cef41ca7411a8406164c1c2bd17c8cac2b000d02
 0000000000000000000935105423ecbb497451dc230f42886399ec
 c3d182c5c07d1d704e009773a18d732d14c68743702175d07363a");
    let block_header_custom1 =
 create_block_custom("00000020051094082528 3d922ebfc7b906c
8f96bc6a98b2d3108010000000000000000ce98
ac444ac2fc3f85ed11b0441981b1fda8d1e5642c8
ee9d2542cc2ee89d0484d44c68043a02170a51c741");
    let block_header_custom2 =
 create_block_custom("0400002a3e0ede0cbfed4883c47eaca3f
d6363dd14afa6eb44f100000000000000000
```

```
0000274928ca8c8b738b7fef8b514893eb80
5251e7a9c879e60b4bc0ddb5ccd7b48b9d84c68043a02172a84512c");

    block_header.append_span(block_header_custom0.span());
    block_header.append_span(block_header_custom1.span());
    block_header.append_span(block_header_custom2.span());
    btc_mirror.submit_block_headers(height, block_header.span());
    btc_mirror.submit_block_headers(height, block_header.span());
}


 fn invalid_header() -> Array<u8> {
     let invalid_header: ByteArray =
@@ -201,23 +218,16 @@ fn wrong_parent_hash_block() -> Array<u8> {
     wrong_parent_hash_bytes.into()
 }

fn create_block_custom(header: ByteArray) -> Array<u8> {
    let header_bytes: ByteArray = from_hex(header);
    header_bytes.into()
}
 /// Generate block #717695 header
 /// All values are little-endian as per Bitcoin protocol
 fn create_block_717695_header() -> Array<u8> {
     // Bitcoin block header with correct little-endian byte order
     // Original mempool data: bits=386635947 (0x170b98ab) stored as
    ab980b17 in little-endian
    let block_header: ByteArray = "04002020"
        + // version (4 bytes)
        "edae5e1bd8a0e007e529fe33d09
            9ebb7a82a06d6d63d0b000000000000000000"
        + // prev hash (32 bytes)
        "f8aec519bcd878c9713dc8153a72
            fd62e3667c5ade70d8d0415584b8528d79ca"
        + // merkle root (32 bytes)
        "0b40d961"
        + // time (4 bytes)
        "ab980b17"

    + // bits (4 bytes) - stored in little-endian, actual value 0x170b98ab
        "3dcc4d5a"; // nonce (4 bytes)
```

```
    let block_header: ByteArray =
"04002020edae5e1bd8a0e007e529fe33d099ebb7a82a06d6d63d0b000
    000000000000000f8aec519bcd878c9713dc8153a72fd62e3667c5ade
    70d8d0415584b8528d79ca0b40d961ab980b173dcc4d5a"; // nonce (4 bytes)
    let header_bytes: ByteArray = from_hex(block_header);
    header_bytes.into()
}
```

and run the test with:

```
scarb test test_submit_block_headers_custom
```

Recommendations:

Compare the total work of the chain being replaced with the total work of the new chain to adhere to Bitcoin consensus rule of following the chain with the most accumulated work.

Fiamma: Resolved with PR-1

Zenith: Verified.

4.3 Low Risk

A total of 2 low risk findings were identified.

[L-1] Incorrect difficulty adjustment boundary check deviates from Bitcoin consensus rules

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- btc_mirror.cairo

Description:

The `_submit_block()` function validates difficulty adjustments during retargeting but uses incorrect boundary checks:

```
assert(
    target < last_target * 4,
    Errors::LESS_THAN_25_PERCENT_DIFFICULTY_RETARGET,
);
```

Bitcoin Core's `CalculateNextWorkRequired` uses $\leqslant$ instead of <, allowing targets exactly equal to 4x the previous target. Additionally, the contract lacks the opposite check (`4 * target` $\geqslant$ `last_target`) to prevent excessive difficulty increases.

This deviation causes the contract to reject valid Bitcoin blocks at the exact 4x boundary.

Recommendations:

Update the comparison operator to match Bitcoin's consensus rules:

```
assert(
    target < last_target * 4, Errors::LESS_THAN_25_PERCENT_DIFFICULTY_
        RETARGET,
);
```

```
assert(
    target ⩽ last_target * 4, Errors::LESS_THAN_25_PERCENT_DIFFICULTY_
        RETARGET,
);
```

Fiamma: Resolved with PR-1

Zenith: Verified.

[L-2] `btc_utils::bits_to_target()` rejects valid block headers whose exponents are 31 or 32

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- btc_utils

Description:

The btc_utils::bits_to_target() function rejects difficulty bits with exponents 31 or 32, which are valid according to bitcoin consensus rules and would fit in a uint256 variable.

Recommendations:

Don't reject block headers whose difficulty bits have exponents of 31 or 32.

Fiamma: Resolved with PR-1

Zenith: Verified.

4.4 Informational

A total of 5 informational findings were identified.

[I-1] Inaccurate Reorg event parameter

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- btc_mirror.cairo

Description:

The `BtcMirror` contract tracks Bitcoin block headers and maintains the canonical chain state. When a block is replaced after being submitted, the contract emits a `Reorg` event containing a count of reorganised blocks. However, this count does not accurately reflect all blocks affected by the reorg.

In `submit_block_headers()`, the `n_reorg` counter is incremented only when a submitted block has a different hash than an existing block at the same height. This occurs in `_submit_block()`:

```
if old_hash ≠ 0 && old_hash ≠ new_hash {
    num_reorged = 1;
}
```

However, when crossing difficulty period boundaries, additional blocks may be invalidated without being counted:

```
// Invalidate any blocks above the new height (handle reorg)
for i in new_height + 1..self.latest_block_height.read() + 1 {
    self.block_height_to_hash.write(i, 0);
}
```

These invalidated blocks represent chain tips that are being reorganised out but are not included in the `n_reorg` count. This results in the `Reorg` event understating the actual number of blocks affected by the reorganisation.

Recommendations:

Consider counting all blocks that are removed from the canonical chain when emitting the `Reorg` event.

Fiamma: Resolved with PR-1

Zenith: Verified.

[I-2] Redundant threshold check after early return logic

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- committee_verifier.cairo

Description:

The `SimpleCommitteeVerifier` contract implements a multi-signature verification system where a threshold number of valid signatures from committee members is required to verify a destination script hash. The `verify_dest_script_hash()` function iterates through provided signatures, validates each one, and maintains a count of valid signatures.

The function implements an optimisation where it returns early as soon as the threshold is met:

```
if valid_count ⩾ threshold {
    self.emit(SignaturesVerified { dest_script_hash });
    return true;
}
```

After the loop completes, there is a redundant expression at line 200:

```
valid_count ⩾ threshold
```

This expression is unnecessary because if the threshold had been met, the function would have already returned `true` within the loop. Therefore, reaching this line guarantees that `valid_count < threshold`.

Recommendations:

Replace the redundant threshold check with an explicit `false` return:

```
valid_count ⩾ threshold
false
```

Fiamma: Resolved with PR-1

Zenith: Verified.

[I-3] Documentation inconsistency regarding owner permissions in burn function

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- bitvm_bridge.cairo

Description:

The `bitvm_bridge` contract implements a `burn()` function that allows users to burn their FIA BTC tokens to initiate Bitcoin withdrawals. The function includes a pause mechanism controlled by the contract owner through the `burn_paused` storage variable.

The implementation contains a comment at line 491 stating "Check if burn is paused (unless caller is unrestricted owner)", which suggests that the owner should have special privileges to bypass the pause check. However, the actual implementation does not include any such bypass logic:

```
// Check if burn is paused (unless caller is unrestricted owner)
if self.burn_paused.read() {
    core::panic_with_felt252(Errors::BURN_PAUSED);
}
```

The function simply checks if burns are paused and reverts for all callers, including the owner. Additionally, the function burns tokens from the caller's own balance via `IFiaBTCDispatcher.burn(caller, value)`, meaning users can only burn their own tokens regardless of who initiates the transaction.

This creates confusion about the intended behaviour and could mislead developers or auditors reviewing the code.

Recommendations:

Update the comment to accurately reflect the implementation:

```
// Check if burn is paused (unless caller is unrestricted owner)
```

```
// Check if burn is paused
if self.burn_paused.read() {
    core::panic_with_felt252(Errors::BURN_PAUSED);
}
```

Fiamma: Resolved with PR-1

Zenith: Verified. Comments were removed.

[I-4] Outdated OpenZeppelin dependencies

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- packages/bridge/src/fia_btc.cairo
- packages/bridge/Scarb.toml

Description:

The bridge package uses OpenZeppelin Cairo contracts version 1.0.0, whilst the latest stable version is 2.0.0. The project is missing out on improvements, bug fixes, and new standardised patterns introduced in newer versions.

For example, the `FiaBTC` contract implements custom decimals by overriding the entire `IERC20Metadata` interface, whilst OpenZeppelin 2.0.0 provides cleaner approaches through the `ImmutableConfig` trait as documented in their ERC20 customisation guide.

Recommendations:

Update OpenZeppelin dependencies to version 2.0.0:

```
[dependencies]
starknet.workspace = true
openzeppelin_access = "1.0.0"
openzeppelin_token = "1.0.0"
openzeppelin_upgrades = "1.0.0"
openzeppelin_security = "1.0.0"
openzeppelin_access = "2.0.0"
openzeppelin_token = "2.0.0"
openzeppelin_upgrades = "2.0.0"
openzeppelin_security = "2.0.0"
```

Fiamma: Resolved in https://github.com/fiamma-chain/starknet-contracts/pull/1

Zenith: Verified

[I-5] `bitvm_bridge::mint()` edge-case allows double-minting

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- bitvm_bridge.cairo

Description:

The function bitvm_bridge:: _process_mint_peg() saves to storage the hash of the transaction id and the transaction index to avoid double-minting from the same transaction:

```
let inclusion_proof_key =
    self.get_inclusion_proof_key(peg.inclusion_proof.tx_id,
    peg.inclusion_proof.tx_index);
// Prevent double-spending by checking if proof was already used
assert(!self.minted.read(inclusion_proof_key),
    Errors::USED_INCLUSION_PROOF);
self.minted.write(inclusion_proof_key, true);
```

The bitcoin protocol constructs the merkle root of the transactions by doubling the last leaf in case the amount of leaves is odd.

This makes it possible to execute bitvm_bridge:: mint() twice with the same raw transaction when:

- the merkle tree contains an odd amount of transactions
- the deposit transaction is the last one in the block

This is possible because the same raw transaction can be passed to the bitvm_bridge:: mint() function with two different indexes.

Recommendations:

In bitvm_bridge:: _process_mint_peg() use only the transaction id as `inclusion_proof_key`.

Fiamma: Resolved with PR-1

Zenith: Verified.