Veridise

Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

catalyst

Bitcoin Prism

Veridise Inc.

December 3, 2024

► **Prepared For:**

Cata Labs
https://catalabs.org/

► **Prepared By:**

Jon Stephens
Benjamin Sepanski
Nicholas Brown

► **Contact Us:**

contact@veridise.com

► **Version History:**

Dec. 3, 2024	V1
Oct. 28, 2024	Initial Draft

Contents

1 Executive Summary

From Oct. 14, 2024 to Oct. 25, 2024, Cata Labs engaged Veridise to conduct a security assessment of their Bitcoin Prism project. The security assessment covered a Bitcoin light client implementation to run on an EVM chain. Veridise conducted the assessment over 4 person-weeks, with 2 security analysts reviewing the project over 2 weeks on commit `c98c043f`. The review strategy involved a tool-assisted analysis of the program source code performed by Veridise security analysts as well as thorough code review.

Project Summary. The Bitcoin Prism project is a light-client for the Bitcoin blockchain and can be used to perform payment verification. It does so by mirroring the *recent* state of bitcoin on-chain that can be queried by users. This state can be updated by anyone so long as the provided chain passes validation that the chain *could be* a valid bitcoin state (e.g. previous block is correct, hash is below target, etc). If two different conflicting chains are submitted, the "heaviest" chain is kept (i.e. the chain with the most work). With these validations, it is therefore likely that the light-client will reflect the true bitcoin state as long as an honest user submits the true state because potential attackers would likely need more hashing power than bitcoin to maintain the heaviest chain. The Bitcoin Prism also provides utilities to help prove information about bitcoin. This includes utilities to retrieve information about the bitcoin state from the light client, the ability to prove that a transaction occurred and utilities to interact with common bitcoin scripts.

Code Assessment. The Bitcoin Prism developers provided the source code of the Bitcoin Prism contracts for the code review. The source code is based on Bitcoin Mirror which is an existing light client implementation for EVM. It contains documentation in the form of READMEs and documentation comments on functions and storage variables. To facilitate the Veridise security analysts understanding of the code, the Bitcoin Prism developers met with the Veridise security analysis to explain the project at a high level.

The source code contained a test suite, which the Veridise security analysts noted covered a variety of success and failure conditions for the various components of the protocol.

Summary of Issues Detected. The security assessment uncovered 9 issues, 2 of which are assessed to be of high or critical severity by the Veridise analysts. Specifically, valid blocks may be erased during a reorg (V-BPR-VUL-001), and Bitcoin transactions aren't properly validated to be legitimate (V-BPR-VUL-002). The Veridise analysts also identified 1 medium-severity issue that the Bitcoin Prism implementation has weaker safety guarantees than Bitcoin (V-BPR-VUL-003) as well as 2 low-severity issues, 3 warnings, and 1 informational finding . The Bitcoin Prism developers have either fixed or responded to all of the issues.

Recommendations. After conducting the assessment of the protocol, the security analysts had a few suggestions to improve the Bitcoin Prism. The Veridise security analysts would recommend performing stricter validation on the inputs to the Bitcoin Prism and documenting outputs that are intended to be error values. They would also recommend that documentation be provided to users so that they are able to evaluate the risks involved when interacting with the project (e.g. how many confirmations should be specified).

Disclaimer. We hope that this report is informative but provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the system is secure in all dimensions. In no event shall Veridise or any of its employees be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the results reported here.

2 Project Dashboard

Table 2.1: Application Summary.

Name	Version	Type	Platform
Bitcoin Prism	`c98c043f`	Solidity	Ethereum

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
Oct. 14–Oct. 25, 2024	Manual & Tools	2	4 person-weeks

Table 2.3: Vulnerability Summary.

Name	Number	Acknowledged	Fixed
Critical-Severity Issues	0	0	0
High-Severity Issues	2	2	2
Medium-Severity Issues	1	1	0
Low-Severity Issues	2	2	1
Warning-Severity Issues	3	2	2
Informational-Severity Issues	1	1	0
TOTAL	9	8	5

Table 2.4: Category Breakdown.

Name	Number
Logic Error	5
Data Validation	3
Denial of Service	1

3 Security Assessment Goals and Scope

3.1 Security Assessment Goals

The engagement was scoped to provide a security assessment of Bitcoin Prism's smart contracts. During the assessment, the security analysts aimed to answer questions such as:

- Can a user successfully submit fake blocks to the Prism?
- Can the project recover if incorrect blocks were previously accepted?
- Will correct block hashes in storage remain intact?
- Is it possible to prove that a transaction happened that didn't actually happened?
- Are inputs properly validated?
- Will the on-chain state mirror the state of Bitcoin with high probability over an extended period?

3.2 Security Assessment Methodology & Scope

Security Assessment Methodology. To address the questions above, the security assessment involved a combination of human experts and automated program analysis & testing tools. In particular, the security assessment was conducted with the aid of the following techniques:

- *Static analysis.* To identify potential common vulnerabilities, security analysts leveraged Veridise's custom smart contract analysis tool Vanguard, as well as the open-source tool Slither. These tools are designed to find instances of common smart contract vulnerabilities, such as reentrancy and uninitialized variables.

Scope. The scope of this security assessment was limited to the `src/` folder of the source code provided by the Bitcoin Prism developers, which contains the smart contract implementation of the Bitcoin Prism.

Methodology. Veridise security analysts inspected the provided tests and read the Bitcoin Prism documentation. They then began a review of the code assisted by static analyzers.

During the security assessment, the Veridise security analysts regularly met with the Bitcoin Prism developers to ask questions about the code.

3.3 Classification of Vulnerabilities

When Veridise security analysts discover a possible security vulnerability, they must estimate its severity by weighing its potential impact against the likelihood that a problem will arise.

The severity of a vulnerability is evaluated according to the Table 3.1.

The likelihood of a vulnerability is evaluated according to the Table 3.2.

Table 3.1: Severity Breakdown.

	Somewhat Bad	Bad	Very Bad	Protocol Breaking
Not Likely	Info	Warning	Low	Medium
Likely	Warning	Low	Medium	High
Very Likely	Low	Medium	High	Critical

Table 3.2: Likelihood Breakdown

Not Likely	A small set of users must make a specific mistake
Likely	Requires a complex series of steps by almost any user(s) - OR - Requires a small set of users to perform an action
Very Likely	Can be easily performed by almost anyone

The impact of a vulnerability is evaluated according to the Table 3.3:

Table 3.3: Impact Breakdown

Somewhat Bad	Inconveniences a small number of users and can be fixed by the user
Bad	Affects a large number of people and can be fixed by the user - OR - Affects a very small number of people and requires aid to fix
Very Bad	Affects a large number of people and requires aid to fix - OR - Disrupts the intended behavior of the protocol for a small group of users through no fault of their own
Protocol Breaking	Disrupts the intended behavior of the protocol for a large group of users through no fault of their own

4 Vulnerability Report

This section presents the vulnerabilities found during the security assessment. For each issue found, the type of the issue, its severity, location in the code base, and its current status (i.e., acknowledged, fixed, etc.) is specified. Table 4.1 summarizes the issues discovered:

Table 4.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-BPR-VUL-001	Incorrect Blocks May be Erased During a . . .	High	Fixed
V-BPR-VUL-002	BitcoinTx not Properly Validated to be . . .	High	Fixed
V-BPR-VUL-003	Prism has Weaker Safety Guarantees than . . .	Medium	Partially Fixed
V-BPR-VUL-004	Single Heavy Block can DoS Prism	Low	Fixed
V-BPR-VUL-005	Missing Transaction Merkle Tree Safety . . .	Low	Partially Fixed
V-BPR-VUL-006	Block Hash of 0 can Always be Proven	Warning	Won't Fix
V-BPR-VUL-007	Re-organized Blocks are Double Counted	Warning	Fixed
V-BPR-VUL-008	Bitcoin Work Calculation is Incorrect	Warning	Fixed
V-BPR-VUL-009	Ordinal Transfer Verification is Restrictive	Info	Acknowledged

4.1 Detailed Description of Issues

4.1.1 V-BPR-VUL-001: Incorrect Blocks May be Erased During a Reorg

Severity	High	**Commit**	c98c043
Type	Logic Error	**Status**	Fixed
File(s)	`BtcPrism.sol`		
Location(s)	submit()		
Confirmed Fix At	`874920a`		

The `submit()` function in `BtcPrism` is used to update the `BtcPrism` with the latest blocks from the Bitcoin chain. The `BtcPrism` implementation stores the last `NUM_BLOCKS` (which is currently set to 2000) blocks in an array called `blockHashes` where each block is indexed by its block number modulo `NUM_BLOCKS` .

When a reorg occurs with a heavier chain that is shorter than the existing chain, the blocks that aren't overwritten by new blocks are reset to zero to prevent invalid blocks from being stored in `blockHashes` . However this loop removes the blocks by indexing the blocks by the block number modulo `MAX_ALLOWED_REORG` (which is currently set to 1000). This means that if the block being erased has a block number such that the block number modulo `NUM_BLOCKS` is greater than `MAX_ALLOWED_REORG`, the block being erased will be a completely different block than the block that should be erased.

```
// erase any block hashes above newHeight, now invalidated.
// (in case we just accepted a shorter, heavier chain.)
for (uint256 i = newHeight + 1; i <= latestBlockHeight; ++i) {
    blockHashes[i % MAX_ALLOWED_REORG] = 0;
}
```

Snippet 4.1: Snippet from `submit()`

Impact Transactions that happened in the blocks that were accidentally erased won't be able to be proven by the `BtcPrism` which could result in significant loss of funds for users. As an example, an honest filler for cross cats could be effectively challenged on an order that was filled during one of the erased blocks.

Recommendation Index by the block number modulo `NUM_BLOCKS` to correctly erase the invalid blocks

Developer Response The developers have implemented the recommendation.

4.1.2 V-BPR-VUL-002: BitcoinTx not Properly Validated to be Legitimate

Severity	High	**Commit**	c98c043
Type	Data Validation	**Status**	Fixed
File(s)	`BtcProof.sol`		
Location(s)	parseBitcoinTx()		
Confirmed Fix At	`147f969`		

The `parseBitcoinTx` function parses a Bitcoin transaction and returns a struct describing the transaction. This function will not revert if the Bitcoin transaction is invalid but instead uses a struct field called `validFormat` to indicate that the transaction was parsed successfully. In the case where the flag is not set, the returned `BitcoinTx` can contain data that was improperly parsed and therefore could allow incorrect data to be accessed as shown below.

```
function parseBitcoinTx(bytes calldata rawTx)
    internal
    pure
    returns (BitcoinTx memory ret)
{
    ...

    // Read transaction outputs
    uint256 nOutputs;
    (nOutputs, offset) = readVarInt(rawTx, offset);
    ret.outputs = new BitcoinTxOut[](nOutputs);
    for (uint256 i = 0; i < nOutputs; ++i) {
        BitcoinTxOut memory txOut;
        txOut.valueSats = Endian.reverse64(
            uint64(bytes8(rawTx[offset:offset += 8]))
        );
        uint256 nOutScriptBytes;
        (nOutScriptBytes, offset) = readVarInt(rawTx, offset);
        txOut.script = rawTx[offset:offset += nOutScriptBytes];
        ret.outputs[i] = txOut;
    }

    // Finally, read locktime, the last four bytes in the tx.
    ret.locktime = Endian.reverse32(
        uint32(bytes4(rawTx[offset:offset += 4]))
    );
    if (offset != rawTx.length) {
        return ret; // Extra data at end of transaction.
    }

    // Parsing complete, sanity checks passed, return success.
    ret.validFormat = true;
    return ret;

    }
}
```

Snippet 4.2: Snippet from `parseBitcoinTx`

When this function is later used by `subvalidate` shown below or the functions that invoke this function, the `validFormat` flag is ignored and the transaction information is used as a trusted source.

```
function subValidate(
    bytes32 blockHash,
    BtcTxProof calldata txProof
) internal pure returns (BitcoinTx memory parsedTx) {
    // 5. Block header to block hash

    bytes calldata proofBlockHeader = txProof.blockHeader;
    bytes32 blockHeaderBlockHash = getBlockHash(proofBlockHeader);
    if (blockHeaderBlockHash != blockHash) revert BlockHashMismatch(
    blockHeaderBlockHash, blockHash);

    // 4. and 3. Transaction ID included in block
    bytes32 txRoot = getTxMerkleRoot(
        txProof.txId,
        txProof.txIndex,
        txProof.txMerkleProof
    );
    bytes32 blockTxRoot = getBlockTxMerkleRoot(proofBlockHeader);
    if (blockTxRoot != txRoot) revert TxMerkleRootMismatch(blockTxRoot, txRoot);

    bytes calldata rawTx = txProof.rawTx;
    // 2. Raw transaction to TxID
    bytes32 rawTxId = getTxID(rawTx);
    if (rawTxId != txProof.txId) revert TxIDMismatch(rawTxId, txProof.txId);

    // Parse raw transaction for further validation.
    return parsedTx = parseBitcoinTx(rawTx);
}
```

Snippet 4.3: Definition of `subValidate`

Impact Any transaction that cannot be parsed by the `parseBitcoinTx` function will therefore not cause the validation to revert (see Bitcoin Transactions with Witnesses do not Parse Correctly for valid transactions that cannot be parsed). This can cause incorrect or 0 data to be used and returned rather than the actual data in the transaction. For cross-cats specifically, this could allow an output to be marked as filled erroneously.

Recommendation `parseBitcoinTx()` should revert if the transaction is invalid, or `subValidate()` should check the value of the flag before returning it.

Developer Response The developers have implemented the recommendation.

4.1.3 V-BPR-VUL-003: Prism has Weaker Safety Guarantees than Bitcoin

Severity	Medium	**Commit**	c98c043
Type	Logic Error	**Status**	Partially Fixed
File(s)	`BtcPrism.sol`		
Location(s)	submitBlock		
Confirmed Fix At	N/A		

The Bitcoin Prism project allows anyone to submit new blocks for inclusion in the light-client chain. To prevent potential tampering, the project allows users to submit a new chain of blocks that can overwrite the previous chain where the "heaviest" chain is kept. Any blocks can be overwritten during this process, meaning a retarget can be included in the chain. This is important because the prism does not precisely model a re-target by calculating the new difficulty. Instead, they enforce that the new target cannot be more than 4 times easier than the current target as seen below.

```
function submitBlock(uint256 blockHeight, bytes calldata blockHeader)
    private
    returns (uint256 numReorged)
{
    ...

    // support once-every-2016-blocks retargeting
    uint256 period = blockHeight / 2016;
    if (blockHeight % 2016 == 0) {
        // Bitcoin enforces a minimum difficulty of 25% of the previous
        // difficulty. Doing the full calculation here does not necessarily
        // add any security. We keep the heaviest chain, not the longest.
        uint256 lastTarget = periodToTarget[period - 1]; // blockHeight > 2016 =>
    blockHeight / 2016 > 1 => fine.
        // ignore difficulty update rules on testnet.
        // Bitcoin testnet has some clown hacks regarding difficulty, see
        // https://blog.lopp.net/the-block-storms-of-bitcoins-testnet/
        if (!isTestnet) {
            if (target >> 2 >= lastTarget) revert DifficultyRetargetLT25();
        }
        periodToTarget[period] = target;
    }

    ...
}
```

Snippet 4.4: Snippet from `SubmitBlock`

Impact As a parallel chain with a lesser difficulty can be accepted, the security guarantees provided from Prism diverge from Bitcoin. This is because in-between bitcoin blocks, it is possible for a user to check-in additional progress made on the less difficult chain. For example, between a retarget block and the first block after the retarget, someone can add blocks with 1/4 the difficulty. Similarly, between the first and second block after a re-target, a chain of 5 or more blocks can be added with 1/4 the difficulty. While the probability of any chain completely

taking over the light client remains extremely low (as doing so would require 1000 blocks on the new chain), the likelihood of small deviations around re-target nodes is increased.

In cross-cats, if a user could cause even a temporary deviation, they could permanently prove that an order has been filled. There would be little risk for such a filler if they were to do so as they could initiate, prove, and fulfill a group of orders all within the same transaction.

Recommendation We would recommend considering one of the following:

1. Increase the number of confirmations needed around retarget boundaries.
2. Implement the same retargeting logic that bitcoin uses.

Developer Response The developers have updated their documentation to recommend multiple confirmations for all orders and have noted that the system may not be safe with one confirmation. This should mitigate the issue in practice but users can be impacted if they don't use the UI or ignore the warnings.

4.1.4 V-BPR-VUL-004: Single Heavy Block can DoS Prism

Severity	Low	**Commit**	c98c043
Type	Denial of Service	**Status**	Fixed
File(s)	`BtcPrism.sol`		
Location(s)	submitBlock		
Confirmed Fix At	`a1f0ff3`		

The Bitcoin Prism project allows anyone to submit new blocks for inclusion in the light-client chain. To prevent a user from forcing the tracked network to diverge from the bitcoin network, the prism project allows previous blocks to be overwritten where the "heaviest" chain is kept. There are some restrictions, however as one can only overwrite the previous 1000 blocks. There are no restrictions on the 1000 blocks that can be overwritten, however, meaning that this can occur while the chain is retargeting. Notably, on Bitcoin retargeting restricts the new target to be within a factor of 4 of the previous target (i.e. can only increase by 4x and reduce by 25%). While Prism enforces the upper bound, the lower bound is not enforced as shown below.

```
function submitBlock(uint256 blockHeight, bytes calldata blockHeader)
    private
    returns (uint256 numReorged)
{
    ...

    // support once-every-2016-blocks retargeting
    uint256 period = blockHeight / 2016;
    if (blockHeight % 2016 == 0) {
        // Bitcoin enforces a minimum difficulty of 25% of the previous
        // difficulty. Doing the full calculation here does not necessarily
        // add any security. We keep the heaviest chain, not the longest.
        uint256 lastTarget = periodToTarget[period - 1]; // blockHeight > 2016 =>
    blockHeight / 2016 > 1 => fine.
        // ignore difficulty update rules on testnet.
        // Bitcoin testnet has some clown hacks regarding difficulty, see
        // https://blog.lopp.net/the-block-storms-of-bitcoins-testnet/
        if (!isTestnet) {
            if (target >> 2 >= lastTarget) revert DifficultyRetargetLT25();
        }
        periodToTarget[period] = target;
    }

    ...
}
```

Snippet 4.5: Snippet from `submitBlock`

Impact Since the heaviest chain will be kept, if someone can submit a block that is 1000 times more difficult than the actual difficulty within 1000 blocks of a retarget, that single block will be considered the heaviest and cannot possibly be a valid bitcoin block. This would effectively break the light client as the bitcoin chain could not overwrite this block.

Recommendation Enforce the restriction on the maximum difficulty increase as well as the minimum difficulty increase.

Developer Response The developers have implemented the above recommendation.

4.1.5 V-BPR-VUL-005: Missing Transaction Merkle Tree Safety Checks

Severity	Low	**Commit**	c98c043
Type	Data Validation	**Status**	Partially Fixed
File(s)	`BtcProof.sol`		
Location(s)	getTxMerkleRoot		
Confirmed Fix At	N/A		

A bitcoin block contains a merkle tree root to succinctly summarize the sequence of transactions executed within the block. To prove that a transaction has been executed, one therefore needs to provide a valid block and a proof that the transaction is included in the block's transaction merkle tree root. Part of the code to validate a merkle tree proof, which we noted is missing some protections implemented in other SPV implementations, is shown below. Such checks include:

1. Checking that the `txIndex` is 0 after the loop terminates to ensure the `siblings` refers to the correct index.
2. Checking if any of the internal nodes correspond to hashes of known transactions. This is to prevent against attacks such as the one described here.

```
function getTxMerkleRoot(
    bytes32 txId,
    uint256 txIndex,
    bytes calldata siblings
) internal pure returns (bytes32) {
    unchecked {

    bytes32 ret = bytes32(Endian.reverse256(uint256(txId)));
    uint256 len = siblings.length / 32;
    for (uint256 i = 0; i < len; ++i) {
        bytes32 s = bytes32(
            Endian.reverse256(
                uint256(bytes32(siblings[i * 32:(i + 1) * 32]))  // i is small.
            )
        );
        ret = doubleSha(
            txIndex & 1 == 0
                ? abi.encodePacked(ret, s)
                : abi.encodePacked(s, ret)
            );
        txIndex = txIndex >> 1;
    }
    return ret;

    }
}
```

Snippet 4.6: Definition of `getTxMerkleRoot`

Impact The above checks are common precautions used to ensure the user is indexing the merkle tree correctly and is referring to real data.

Recommendation We would recommend implementing similar checks to those described above.

Developer Response The developers have fixed the first part of this issue. They have also placed limits on the size of trades in the short term to reduce the economic incentive of such an attack. They plan to address the second issue in the future since it requires more substantial changes.

4.1.6 V-BPR-VUL-006: Block Hash of 0 can Always be Proven

Severity	Warning	**Commit**	c98c043
Type	Data Validation	**Status**	Won't Fix
File(s)	`BtcPrism.sol`, `BtcTxVerifier.sol`, `BitcoinOracle.sol`		
Location(s)	Multiple		
Confirmed Fix At	N/A		

The Bitcoin Prism light client allows users to retrieve the block hash associated with a given block number as long as the requested hash is less than or equal to the latest block tracked by prism. Notably, however, if the requested block number is too old, then 0 will be returned rather than the requested hash.

```
function getBlockHash(uint256 blockNum) public view returns (bytes32) {
    require(blockNum <= latestBlockHeight, "Block not yet submitted");
    if (blockNum < latestBlockHeight - MAX_ALLOWED_REORG) {
        return 0;
    }
    return blockHashes[blockNum % NUM_BLOCKS];
}
```

Snippet 4.7: Definition of `getBlockHash`

Impact If the return value of `getBlockHash` is not checked to see if the output is 0, a user can always prove that a block with the zero hash is included in the blockchain. This potentially allows incorrect information to be proven. Note this holds for the `BtcTxVerifier` and `BitcoinOracle`.

Recommendation Consider reverting if the returned hash is 0.

Developer Response The developers have indicated that they do not plan to fix this issue.

4.1.7 V-BPR-VUL-007: Re-organized Blocks are Double Counted

Severity	Warning	Commit	c98c043
Type	Logic Error	Status	Fixed
File(s)	BtcPrism.sol		
Location(s)	submit		
Confirmed Fix At	54add79		

When a user submits new blocks for inclusion in Prism, some blocks can be overwritten if the new chain is heavier. In this case, the number of re-organized or overwritten blocks will be reported to the user. The calculation that counts the number of re-organized blocks, however, appears to be incorrect as the number of reorganized blocks is not counted. Rather, as shown below, for each re-organized block the distance between the block and the old latest block is accumulated. If the new chain contains multiple blocks, some re-organized blocks will therefore be double-counted.

```
function submit(uint256 blockHeight, bytes calldata blockHeaders) public {
    ...

    for (uint256 i = 0; i < numHeaders; ++i) {
        // unchecked: This might overflow if numheaders = type(uint256).max - type(
    uint256).max/2016. But that is such a massive number
        // that it is not going to overflow.
        uint256 blockNum = blockHeight + i;
        nReorg += submitBlock(blockNum, blockHeaders[80 * i:80 * (i + 1)]); //
    unchecked: Overflows if blockHeaders.length == type(uint256).max.
    }

    ...

    if (nReorg > 0) {
        emit Reorg(nReorg, oldTip, newTip);
    }
}

function submitBlock(uint256 blockHeight, bytes calldata blockHeader)
    private
    returns (uint256 numReorged)
{
    ...

    // optimistically save the block hash
    // we'll revert if the header turns out to be invalid
    if (blockHeight <= latestBlockHeight) {
        // if we're overwriting a non-zero block hash, that block is reorged
        numReorged = latestBlockHeight - blockHeight;
    }

    ...
}
```

Snippet 4.8: Logic showing how the number of re-organized blocks is computed

Impact The number of re-organized blocks reported to users may be inaccurate.

Recommendation We believe that rather than accumulating the return value from `submitBlock`, the maximum value should be selected.

Developer Response The developers have simplified the logic by computing the number of re-organized blocks using the difference between the chain heights.

4.1.8 V-BPR-VUL-008: Bitcoin Work Calculation is Incorrect

Severity	Warning	**Commit**	c98c043
Type	Logic Error	**Status**	Fixed
File(s)	`BtcPrism.sol`		
Location(s)	getWorkInPeriod		
Confirmed Fix At	`cdb5905`		

The prism project allows users to submit an alternative chain of blocks which can overwrite the current chain if it "heavier." The heaviness of the block is determined by calculating the amount of work that was performed in each chain.

```
function getWorkInPeriod(uint256 period, uint256 height)
    private
    view
    returns (uint256)
{
    unchecked {

    uint256 target = periodToTarget[period];
    uint256 workPerBlock = (2**256 - 1) / target;

    // unchecked: period is not raw from input but parsed as newHeight/2016.
    // as such, we can multiply it by 2016.
    uint256 numBlocks = height - (period * 2016) + 1;
    require(numBlocks >= 1 && numBlocks <= 2016);

    return numBlocks * workPerBlock;

    }
}
```

Snippet 4.9: Definition from `getWorkInPeriod`

Prism calculates the work using the `getWorkInPeriod` function shown above, but this does not match the definition of work used by bitcoin, which is shown below.

```
arith_uint256 GetBlockProof(const CBlockIndex& block)
{
    ...
    // We need to compute 2**256 / (bnTarget+1), but we can't represent 2**256
    // as it's too large for an arith_uint256. However, as 2**256 is at least as
    large
    // as bnTarget+1, it is equal to ((2**256 - bnTarget - 1) / (bnTarget+1)) + 1,
    // or ~bnTarget / (bnTarget+1) + 1.
    return (~bnTarget / (bnTarget + 1)) + 1;
}
```

Snippet 4.10: Work calculation used by Bitcoin

Recommendation While it is unlikely that this will have a significant impact when determining the heaviness of a chain, we would recommend using Bitcoin's formula to avoid any discrepencies.

Developer Response The developers have applied the above recommendation.

4.1.9 V-BPR-VUL-009: Ordinal Transfer Verification is Restrictive

Severity	Info	Commit	c98c043
Type	Logic Error	Status	Acknowledged
File(s)	`BtcProof.sol`		
Location(s)	validateOrdinalTransfer		
Confirmed Fix At	N/A		

The Bitcoin Prism project includes functionality to validate the transfer of ordinals from one user to another. It does so by allowing a user to specify the UTXO of the desired ordinal and the value necessary to transfer the requested ordinals. The oracle then validates that the transaction's first input corresponds to the specified UTXO, the transaction's first output has the expected script and that the value of the first output meets or exceeds the input value. Note that this works because ordinals are transferred in a first-in-first-out order and therefore if the value sent meets or exceeds the requested value, the desired ordinals must be included. This validation method, however, is restrictive as it essentially requires that a user transfer *all* of their ordinals with a value less than the the desired ordinal.

```
function validateOrdinalTransfer(
    bytes32 blockHash,
    BtcTxProof calldata txProof,
    uint256 txInId,
    uint32 txInPrevTxIndex,
    bytes calldata outputScript,
    uint256 satoshisExpected
) internal pure returns (bool) {
    // 1. Finally, validate raw transaction correctly transfers the ordinal(s).
    // Parse transaction
    BitcoinTx memory parsedTx = subValidate(blockHash, txProof);
    BitcoinTxIn memory txInput = parsedTx.inputs[0];
    // Check if correct input transaction is used.
    if (txInId != txInput.prevTxID) revert InvalidTxInHash(txInId, txInput.prevTxID);
    // Check if correct index of that transaction is used.
    if (txInPrevTxIndex != txInput.prevTxIndex) revert InvalidTxInIndex(
    txInPrevTxIndex, txInput.prevTxIndex);

    BitcoinTxOut memory txo = parsedTx.outputs[0];
    // if the length are less than 32, then use bytes32 to compare.
    if (!compareScriptsCM(outputScript, txo.script)) revert ScriptMismatch(
    outputScript, txo.script);

    // We allow for sending more because of the dust limit which may cause problems.
    if (txo.valueSats < satoshisExpected) revert AmountMismatch(txo.valueSats,
    satoshisExpected);

    // We've verified that blockHash contains a transaction with correct script
    // that sends at least satoshisExpected to the given hash.
    return true;
}
```

Snippet 4.11: Definition of `validateOrdinalTransfer`

Impact Ordinal transfers are often split so that the user can retain ownership of ordinals that should not be included in the transfer. They cannot do so with the above API unless they use multiple transactions.

Recommendation Consider expanding on this API so that users can more accurately transfer specific ordinals

Developer Response The developers have acknowledged this issue.