Veridise

Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

Boundless Transceiver

Veridise Inc.

Sep. 15, 2025

► **Prepared For:**

Wormhole
https://wormhole.com/

► **Prepared By:**

Tyler Diamond
Kostas Ferles

► **Contact Us:**

contact@veridise.com

► **Version History:**

Sep. 15, 2025	V2
Aug. 18, 2025	V1
Aug. 14, 2025	Initial Draft

Contents

1 Executive Summary

From Aug. 11, 2025 to Aug. 13, 2025, Wormhole engaged Veridise to conduct a security assessment of their Boundless Transceiver. The security assessment covered two smart contracts surrounding a system of storing Ethereum beacon chain block roots. Veridise conducted the assessment over 6 person-days, with 2 security analysts reviewing the project over 3 days at commit `c7651b0`. The review strategy involved a tool-assisted analysis of the program source code performed by Veridise security analysts as well as a thorough code review.

Project Summary. The security assessment covered the `BeaconEmitter`, `BoundlessReceiver` and their respective dependencies. The former contract will read Beacon chain block roots for a requested slot from the Beacon roots address (as specified in EIP-4788), and proceed to multicast the slot along with the returned root to Wormhole.

The `BoundlessReceiver` contract is deployed on another chain and stores the configured source chain's block roots. This contract considers two sources of truth for confirmed roots. The first is consuming Wormhole messages emitted by the `BeaconEmitter` of the source chain. The second source consists of RiscZero proofs. The corresponding zkVM program proves that the derivation of the proposed new block root is correct. Note that the security analysts assumed that the zkVM implementation is correct and protects against known attacks on Ethereum's consensus algorithm, e.g., balance drift attacks.

The `BoundlessReceiver` contract stores which methods have attested to the validity of every block root. Users of the `BoundlessReceiver` contract can then query the root for a specific slot and also check whether the returned root has been attested to by either or both validation methods.

Code Assessment. The Boundless Transceiver developers provided the source code of the Boundless Transceiver contracts for the code review. The source code appears to be mostly original code written by the Boundless Transceiver developers. It contains some documentation in the form of READMEs and documentation comments on functions and storage variables. The analysts note that not all functions were documented, and particularly the `BoundlessReceiver` had no documentation. To facilitate the Veridise security analysts' understanding of the code, the Boundless Transceiver developers provided the repository for the corresponding zkVM implementation.

The source code contained a test suite, which the Veridise security analysts noted contained mostly positive test cases of the contracts. The analysts also noted that the `BeaconEmitterTest` mocked several of its dependencies.

Summary of Issues Detected. The security assessment uncovered 7 issues, 3 of which are assessed to be of low severity by the Veridise analysts. Specifically, one issue (V-WMH-VUL-003) demonstrates that the protocol might be susceptible to long-range attacks. Another low severity issue (V-WMH-VUL-002) shows that the `BeaconEmitter` might lock funds if users send the

wrong fee. The Veridise analysts also identified 3 warnings and 1 informational finding. The Boundless Transceiver developers have been notified about all issues and are planning to fix them promptly.

Recommendations. After conducting the assessment of the protocol, the security analysts had a few suggestions to improve the Boundless Transceiver.

Improve Testing. As mentioned above, the Veridise analysts assessed portions of the protocol to be insufficiently tested. The Veridise suggest introducing more tests (especially negative ones) in the test suite and reduce the number of mocked components in the tests. This will ensure that future changes of the protocol will not introduce regressions, increase confidence on the security of the protocol, and that external dependencies are not mocked inaccurately (see V-WMH-VUL-001).

Improve Documentation. As mentioned above, portions of the codebase are completely undocumented. Because of this, the reason for certain invariants are unclear, such as the way `permissibleTimespan` is used. In addition to commenting the code, we recommend explicitly documenting the security assumptions that the two main smart contracts rely on.

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
Boundless Transceiver	`c7651b0`	Solidity	Ethereum

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
Aug. 11–Aug. 13, 2025	Manual & Tools	2	6 person-days

Table 2.3: Vulnerability Summary.

Name	Number	Acknowledged	Fixed
Critical-Severity Issues	0	0	0
High-Severity Issues	0	0	0
Medium-Severity Issues	0	0	0
Low-Severity Issues	3	3	2
Warning-Severity Issues	3	3	3
Informational-Severity Issues	1	1	1
TOTAL	7	7	6

Table 2.4: Category Breakdown.

Name	Number
Maintainability	4
Logic Error	2
Data Validation	1

3 Security Assessment Goals and Scope

3.1 Security Assessment Goals

The engagement was scoped to provide a security assessment of Boundless Transceiver's source code. During the assessment, the security analysts aimed to answer questions such as:

- Does the source code deviate from its intended specification?
- Is the protocol protected against common attacks in Ethereum's consensus algorithm (e.g., long-range attacks)?
- Are external dependencies (e.g., Wormhole SDK, RiscZero verifier) being used properly?
- Are any common smart contract vulnerabilities present in the codebase?
- Is the EVM modeled properly by the protocol?

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. 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 is limited to the `src/` folder of the source code provided by the Boundless Transceiver developers, which contains the smart contract implementation of the Boundless Transceiver.

Methodology. Veridise security analysts inspected the provided tests and read the Boundless Transceiver documentation, along with the corresponding zkvm implementation documentation. They then began a review of the code assisted by both static analyzers and automated testing.

During the security assessment, the Veridise security analysts communicated regularly with the Boundless Transceiver developers over a shared channel 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 Trust Model

4.1 Operational Assumptions

In addition to assuming that any out-of-scope components behave correctly, Veridise analysts assumed the following properties held when modeling security for Boundless Transceiver.

- The protocol is aware and protects against any limitations of the zkVM guest program used to generate the proofs.

4.2 Privileged Roles

Roles. This section describes in detail the specific roles present in the system, and the actions each role is trusted to perform. The roles are grouped based on two characteristics: privilege-level and time-sensitivity. *Highly-privileged* roles may have a critical impact on the protocol if compromised, Time-sensitive *emergency* roles may be required to perform actions quickly based on real-time monitoring, while *non-emergency* roles perform actions like deployments and configurations which can be planned several hours or days in advance.

During the review, Veridise analysts assumed that the role operators perform their responsibilities as intended. Protocol exploits relying on the below roles acting outside of their privileged scope are considered outside of scope.

- Highly-privileged, emergency roles:
  - `BoundlessReceiver.ADMIN_ROLE` can manually transfer the protocol to a new state, update the image ID of the ZKVM application, and update the permissible timespan.
- Highly-privileged, non-emergency roles:
  - `BoundlessReceiver.DEFAULT_ADMIN_ROLE` can grant roles (including `ADMIN_ROLE`) to users.

Operational Recommendations. Highly-privileged, non-emergency operations should be operated by a multi-sig contract or decentralized governance system. These operations should be guarded by a timelock to ensure there is enough time for incident response. It is also recommended that `BoundlessReceiver` inherits from OZ `AccessControlDefaultAdminRules`, which automatically introduces appropriate timelocks. Highly-privileged, emergency operations should be tested in example scenarios to ensure the role operators are available and ready to respond when necessary.

Full validation of operational security practices is beyond the scope of this review. Users of the protocol should ensure they are confident that the operators of privileged keys are following best practices such as:

- Never storing a protocol key in plaintext, on a regularly used phone, laptop, or device, or relying on a custom solution for key management.
- Using separate keys for each separate function.
- Storing multi-sig keys in a diverse set of key management software/hardware services and geographic locations.
- Enabling 2FA for key management accounts. SMS should *not* be used for 2FA, nor should any account which uses SMS for 2FA. Authentication apps or hardware are preferred.
- Validating that no party has control over multiple multi-sig keys.
- Performing regularly scheduled key rotations for high-frequency operations.
- Securely storing physical, non-digital backups for critical keys.
- Actively monitoring for unexpected invocation of critical operations and/or deployed attack contracts.
- Regularly drilling responses to situations requiring emergency response such as pausing/unpausing.

5 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 5.1 summarizes the issues discovered:

Table 5.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-WMH-VUL-001	Wrong Ethereum Beacon genesis timestamp	Low	Fixed
V-WMH-VUL-002	Value greater than the Wormhole fee is lost . . .	Low	Fixed
V-WMH-VUL-003	BoundlessReceiver is susceptible to long- . . .	Low	Partially Fixed
V-WMH-VUL-004	Incorrect TWO_OF_TWO_FLAG	Warning	Fixed
V-WMH-VUL-005	Hardcoded CONSISTENCY_LEVEL may . . .	Warning	Fixed
V-WMH-VUL-006	Inconsistencies in Beacon constants	Warning	Fixed
V-WMH-VUL-007	Maintainability Improvements	Info	Fixed

5.1 Detailed Description of Issues

5.1.1 V-WMH-VUL-001: Wrong Ethereum Beacon genesis timestamp

Severity	Low	Commit	cb2aa61
Type	Logic Error	Status	Fixed
Location(s)	src/lib/Beacon.sol:12-13		
Confirmed Fix At	cbd1375		

The constant `ETHEREUM_GENESIS_BEACON_BLOCK_TIMESTAMP` is set to **1_606_824_000**, which is the **minimum** genesis time, not the **actual** Beacon Chain genesis time. The Beacon Chain actually started at **1_606_824_023** (Dec-01-2020 12:00:23 UTC).

Impact This can affect the codebase in the following two ways:

1. **findBlockRoot failures:**
   EIP-4788's ring buffer validates (parent_beacon_block_root, timestamp) pairs. If this constant is used as the first parameter of `findBlockRoot` (see snippet below), lookups can miss the stored entry and revert with NoBlockRootFound().
2. **Transition window expanded by 23s:**
   `_permissibleTransition` (see snippet below) uses `Beacon.ETHEREUM_GENESIS_BEACON_BLOCK_TIMESTAMP` to calculate the timestamp of `post.finalizedCheckpoint.epoch`.
   Using a genesis 23s earlier decreases the value of `transitionTimespan` by 23s, effectively permitting slightly **later** checkpoints than initially intended.

Relevant code snippets:

```
function findBlockRoot(
    uint256 genesisBlockTimestamp,
    uint64 slot
) public view returns (bytes32 blockRoot) {
    uint256 currBlockTimestamp = genesisBlockTimestamp + ((slot + 1) * 12);

    uint256 earliestBlockTimestamp = block.timestamp -
        (BEACON_ROOTS_HISTORY_BUFFER_LENGTH * 12);
    if (currBlockTimestamp < earliestBlockTimestamp) {
        revert TimestampOutOfRange();
    }

    while (currBlockTimestamp <= block.timestamp) {
        (bool success, bytes memory result) = BEACON_ROOTS_ADDRESS
            .staticcall(abi.encode(currBlockTimestamp));
        if (success && result.length > 0) {
            return abi.decode(result, (bytes32));
        }

        unchecked {
            currBlockTimestamp += BLOCK_SPEED;
        }
    }
```

```
24
25      revert NoBlockRootFound();
26  }
```

```
1   function _permissibleTransition(
2       ConsensusState memory pre,
3       ConsensusState memory post
4   )
5       internal
6       view
7       returns (bool)
8   {
9       uint256 transitionTimespan = block.timestamp
10          - Beacon.epochTimestamp(Beacon.ETHEREUM_GENESIS_BEACON_BLOCK_TIMESTAMP, post.
    finalizedCheckpoint.epoch);
11      return transitionTimespan <= uint256(permissibleTimespan);
12  }
```

Recommendation Use the actual mainnet genesis time: change constant to 1_606_824_023.

Developer Response The developers changed the constant to the correct genesis timestamp.

5.1.2 V-WMH-VUL-002: Value greater than the Wormhole fee is lost to contract

Severity	Low	**Commit**	cb2aa61
Type	Data Validation	**Status**	Fixed
Location(s)	`src/BeaconEmitter.sol:35-41`		
Confirmed Fix At	d6d0ff0		

The `emitForSlot()` function calculates the required fee from Wormhole (`wormholeFee`), and then only sends `wormholeFee` to the Wormhole core contract in the `publishMessage()` call. The rest of the value sent in this call is retained by the contract. Since this contract has no function to claim this leftover value, this ETH is permanently lost to the contract.

Impact ETH that is not sent along to Wormhole is lost to the contract.

Recommendation Return the leftover value to the sender.

Developer Response The developers now send the entire `msg.value` in the `publishMessage()` call.

Updated Veridise Response Any value that is not exactly equal to the `wormholeFee` will cause a revert and cost the sender gas. This should be clearly documented for callers of this function and avoided in any frontend implementations.

5.1.3 V-WMH-VUL-003: BoundlessReceiver is susceptible to long-range attacks

Severity	Low	**Commit**	cb2aa61
Type	Logic Error	**Status**	Partially Fixed
Location(s)	`src/BoundlessReceiver.sol:171-182`		
Confirmed Fix At	78a303b		

Ethereum's Proof-of-Stake (PoS) model relies on the concept of weak subjectivity. At a high level, the PoS model drives block production by randomly selecting a validator out of the pool of all validators and granting them the right to propose a block for a given slot. After this block has been proposed, a subset of validators on the network will validate and attest to the block, essentially "voting" that this should be the next block in the canonical chain. Over the span of 32 blocks, referred to as an epoch, all validators should have voted on a block. Therefore, in order for a node to determine if a block should be added to its local view of the canonical chain, it must have access to the set of validators.

Long-range attacks may occur in the PoS model if a significant (but not majority) portion of validators collude. These attackers may build a private, alternate chain and withdraw their stake on the real chain. They then attempt to reorganize previously validated blocks and will suffer no consequences if their attack fails. In order to protect against this, at defined block intervals a node will store the block as the **weak subjectivity checkpoint**. Nodes will no longer accept any alternative blocks for this block, and therefore long-range reorganizations are prevented. Notably, this length of time (the **weak subjectivity period**) is shorter than the time it takes for validators to withdraw their stake.

An issue arises when a node has been offline for longer than the weak subjectivity period. With outdated validator information, and without any outside information, a node will be unable to determine the true canonical chain. Normal Ethereum nodes utilize outside information to solve this, such as requesting the latest weak subjectivity checkpoint from multiple nodes, asking a trusted node, or updating their node software that embeds the latest known checkpoints.

For this project, we can consider the `BoundlessReceiver` as a light node. Just like light nodes, updates to the `currentState` cannot be made after the weak subjectivity period has elapsed without outside intervention. For the `BoundlessReceiver`, the outside intervention takes the form of an admin manually setting the latest `currentState`.

To protect against the aforementioned long-range attacks, the contract will enforce that the `currentState` cannot progress for a period longer than `permissibleTimespan`, which should correspond to the weak subjectivity period.

```
function _permissibleTransition(
    ConsensusState memory pre,
    ConsensusState memory post
)
    internal
    view
    returns (bool)
{
    uint256 transitionTimespan = block.timestamp
        - Beacon.epochTimestamp(Beacon.ETHEREUM_GENESIS_BEACON_BLOCK_TIMESTAMP, post.
    finalizedCheckpoint.epoch);
```

```
11      return transitionTimespan <= uint256(permissibleTimespan);
12  }
```

Snippet 5.1: Snippet from `BoundlessReceiver.sol`

However, this incorrectly checks whether the *new* state update epoch is within the weak subjectivity period, instead of checking if the transition *from* the current latest state is within the period.

Impact If the corresponding zkVM program allows an arbitrary number of epochs to be proven, then the weak subjectivity period checks that nodes normally check is bypassed. This enables a portion of the validator set that exists at the `currentState` to possibly mount a long range attack and store a non-canonical state root in the contract.

Recommendation Check that the `pre/currentState` is within the weak subjectivity period.

Developer Response The developers now check that the `currentState`'s epoch is within the `permissibleTimestamp`.

Updated Veridise Response This change sufficiently checks that the weak subjectivity period is observed. However, the new state is not checked against the `block.timestamp` to ensure it is not from the future.

5.1.4 V-WMH-VUL-004: Incorrect TWO_OF_TWO_FLAG

Severity	Warning	**Commit**	cb2aa61
Type	Maintainability	**Status**	Fixed
Location(s)	`src/BoundlessReceiver.sol:17`		
Confirmed Fix At	73f8b24		

The `TWO_OF_TWO_FLAG` is supposed to represent both the `WORMHOLE_FLAG` and `BOUNDLESS_FLAG`. As can be seen below, attestations are set using bitwise OR with `(1 << flag)`

```
uint16 public constant BOUNDLESS_FLAG = 0;
uint16 public constant WORMHOLE_FLAG = 1;
uint16 public constant TWO_OF_TWO_FLAG = BOUNDLESS_FLAG | WORMHOLE_FLAG;

// ...

function _confirm(uint16 confirmations, uint16 flag) internal pure returns (uint16) {
    return uint16(confirmations | (1 << flag));
}
```

Snippet 5.2: Snippet from `BoundlessReceiver.sol`

This means the `TWO_OF_TWO_FLAG` does not correctly represent both flags, and should instead be set as `(1 << BOUNDLESS_FLAG) | (1 << WORMHOLE_FLAG)`.

Impact If a consumer of this contract utilizes this constant, then they will mistakenly only validate that the `BOUNDLESS_FLAG` has been set in the corresponding attestation.

Recommendation Set the `TWO_OF_TWO_FLAG` to `(1 << BOUNDLESS_FLAG) | (1 << WORMHOLE_FLAG)`

Developer Response The developers have fixed the constant according to the above recommendation.

5.1.5 V-WMH-VUL-005: Hardcoded CONSISTENCY_LEVEL may not be valid on other networks

Severity	Warning	**Commit**	cb2aa61
Type	Maintainability	**Status**	Fixed
Location(s)	`src/BeaconEmitter.sol:19`		
Confirmed Fix At	3628959		

The `CONSISTENCY_LEVEL` constant is set to 0, and is used as a parameter to the Wormhole `publishMessage()` function. According to Wormhole's documentation, in Ethereum this is an unknown value and is treated as requiring finalization. This is safe, and currently all other EVM-based chains use `0` to represent a finalization requirement. However, this is not guaranteed in the future.

Impact If an EVM chain is released where Wormhole utilizes the `0` value for a level less permanent than finalization, then the security of values attested by this emitter are impacted.

Recommendation Make this an `immutable` variable that is set at construction time.

Developer Response The developers have been notified of the issue but have yet to respond with acknowledgement or fixes.

5.1.6 V-WMH-VUL-006: Inconsistencies in Beacon constants

Severity	Warning	**Commit**	cb2aa61
Type	Maintainability	**Status**	Fixed
Location(s)	`src/` ▶ `BeaconEmitter.sol` ▶ `lib/Beacon.sol:39`		
Confirmed Fix At	3628959		

The `findBlockRoot()` function takes a parameter `genesisBlockTimestamp`. This value is used to calculate the timestamp of the requested `slot` in order to query the EIP-4788 contract. However, this value is already defined as the `ETHEREUM_GENESIS_BEACON_BLOCK_TIMESTAMP`. Since the only consumer of this function is the `BeaconEmitter`, which should be utilizing the same constant, there is no reason this should be a parameter to the function.

Additionally, the `BeaconEmitter` takes in `genesisBlockTimestamp` as a constructor argument, and this is the value it uses on calls to `findBlockRoot()`. This redundant definition and unnecessary parameterization may cause confusion. Since the other constants, such as `BLOCK_TIME` and the `BEACON_ROOTS_ADDRESS`, are specific to Ethereum; the code therefore already relies on chain-specific constants.

Impact Maintainability of the codebase can become more difficult, requiring updates across multiple places for different deployments.

Recommendation The security analysts recommend one of two fixes:

1. Remove the `genesisBlockTimestamp` from `findBlockRoot()` in `Beacon` and just utilize `ETHEREUM_GENESIS_BEACON_BLOCK_TIMESTAMP`. Additionally, remove the `GENESIS_BLOCK_TIMESTAMP` from `BeaconEmitter`. Lastly, change the constant to specify this is for mainnet deployments such as: `ETHEREUM_MAINNET_GENESIS_BEACON_BLOCK_TIMESTAMP`.
2. Remove all mainnet-specific constants from `Beacon` and introduce an auxiliary struct that stores all chain parameters (e.g., genesis timestamp, EIP-4788 contract, block speed, etc.). Then refactor all functions in `Beacon` to operate over the auxiliary data structure instead of relying on hardcoded constants.

Developer Response The developers introduced an auxiliary data structure to model Beacon's configuration parameters.

5.1.7 V-WMH-VUL-007: Maintainability Improvements

Severity	Info	**Commit**	cb2aa61
Type	Maintainability	**Status**	Fixed
Location(s)	`src/` ▶ `BoundlessReceiver.sol` ▶ `lib/Beacon.sol`		
Confirmed Fix At	a86ebfa		

The maintainability of the code may be improved by changes in the following locations:

1. `Beacon.sol`:
    a) `L42-45`: The `BLOCK_SPEED` should be used instead of the hardcoded `12`.
2. `BoundlessReceiver.sol`
    a) `L133`: The `attestation` variable should be a `memory` type instead of `storage` to avoid accidental storage changes.
    b) `L205`: The `remainder` is masked with a bitwise AND with the `confirmationLevel`, and therefore should be compared with a strict equality (==), as it can never be greater than.
    c) `updateImageID()`: We recommend emitting an event when this function is called so those that are submitting zk proofs do not waste gas submitting a proof with an incorrect `imageID`.

Impact Issues may arise in the future from inconsistencies in the codebase. Additionally, the recommended changes will improve the ability to understand the code.

Recommendation Implement the recommendations.

Developer Response The developers have fixed all maintainability issues.

Glossary

RiscZero RiscZero enables developers to create zero-knowledge applications by compiling vanilla Rust code into the RISC-V instruction set for consumption in their zkVM implementation. See https://risczero.com/ . 1

smart contract A self-executing contract with the terms directly written into code. Hosted on a blockchain, it automatically enforces and executes the terms of an agreement between buyer and seller. Smart contracts are transparent, tamper-proof, and eliminate the need for intermediaries, making transactions more efficient and secure. 1

zero-knowledge circuit A cryptographic construct that allows a prover to demonstrate to a verifier that a certain statement is true, without revealing any specific information about the statement itself. See https://en.wikipedia.org/wiki/Zero-knowledge_proof for more. 18

zkVM A general-purpose zero-knowledge circuit that implements proving the execution of a virtual machine. This enables general purpose programs to prove their execution to outside observers, without the manual constraint writing usually associated with zero-knowledge circuit development . 1, 18