Axiom

Security Assessment

October 12, 2023

Prepared for:

Yi Sun

Axiom

Prepared by:

Allen Roh and Mohit Sharma

Zellic Inc.

Contents

About Zellic

Zellic was founded in 2020 by a team of blockchain specialists with more than a decade of combined industry experience. We are leading experts in smart contracts and Web3 development, cryptography, web security, and reverse engineering. Before Zellic, we founded perfect blue, the top competitive hacking team in the world. Since then, our team has won countless cybersecurity contests and blockchain security events.

Zellic aims to treat clients on a case-by-case basis and to consider their individual, unique concerns and business needs. Our goal is to see the long-term success of our partners rather than simply provide a list of present security issues. Similarly, we strive to adapt to our partners' timelines and to be as available as possible. To keep up with our latest endeavors and research, check out our website zellic.io or follow @zellic_io on Twitter. If you are interested in partnering with Zellic, please contact us at hello@zellic.io.

About KALOS

KALOS is a flagship service of HAECHI LABS, providing blockchain wallets and security audits since 2018.

We bring together the best experts to make the Web3 space safer for everyone. Our team consists of security researchers with various expertise — smart contract, blockchain, cryptography, web security, reverse engineering, and binary analysis. Their skills have lead to multiple strong performances in reputable cybersecurity competitions over the past few years.

Over the course of the last five years, we have secured nearly $60B crypto assets over 400 projects of various types such as mainnets, DeFi protocols, NFT services, P2E games, and bridges. Our expertise was recognized by the Samsung Electronics Startup Incubation Program, and we have also received technology grants from the Ethereum Foundation and the Ethereum Community Fund.

Our audit process is customer focused — our security researchers communicate with the team on a regular basis, sharing key vulnerabilities as soon as they are discovered. With our expertise and our personalized approach for each client, we believe that our security audits will be a great addition for your project.

Our website with our profiles and recent research is at kalos.xyz. If you are interested in getting an audit with us, please send us an email at audit@kalos.xyz.

1 Executive Summary

Zellic and KALOS conducted a security assessment for Axiom from September 4th to October 2nd, 2023. During this engagement, we reviewed Axiom's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.1 Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic, KALOS, and the client. In this assessment, we sought to answer the following questions:

- Do the circuits follow the appropriate specification?
- Are the circuits constrained properly?
- Are the witness assignments done correctly?

1.2 Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Front-end components
- Infrastructure relating to the project
- Key custody

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

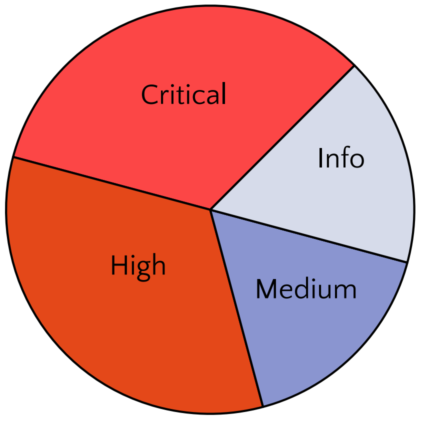
1.3 Results

During our assessment on the scoped Axiom circuits, we discovered six findings. Two critical issues were found. Two were of high impact, one was of medium impact, and the remaining finding was informational in nature.

Additionally, we recorded our notes and observations from the assessment for Axiom's benefit in the Discussion section (4) at the end of the document.

Breakdown of Finding Impacts

Impact Level	Count
Critical	2
High	2
Medium	1
Low	0
Informational	1



2 Introduction

2.1 About Axiom

Axiom is a ZK coprocessor for Ethereum that provides smart contracts trustless access to all on-chain data and arbitrary expressive compute over it. Developers can make queries into Axiom and trustlessly use the ZK-verified results on chain in their smart contracts.

2.2 Methodology

During a security assessment, Zellic and KALOS work through various testing methods along with a manual review. In some cases for a ZKP circuit, we also provide some proofs for soundness. The majority of the time is spent on a manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, we focus primarily on the following classes of security and reliability issues:

Underconstrained circuits. The most common type of vulnerability in a ZKP circuit is not adding sufficient constraints to the system. This leads to proofs generated with incorrect witnesses in terms of the specification of the project being accepted by the ZKP verifier. We manually check that the set of constraints satisfies soundness, enough to remove all such possibilities, and in some cases, provide a proof of the fact.

Overconstrained circuits. While rare, it is possible that a circuit is overconstrained. In this case, appropriately assigning witness will become impossible, leading to a vulnerability. To prevent this, we manually check that the constraint system is set up with completeness so that the proofs generated with the correct set of witnesses indeed pass the ZKP verification.

Missing range checks. This is a popular type of an underconstrained circuit vulnerability. Due to the usage of field arithmetic, overflow checks and range checks serve a huge purpose to build applications that work over the integers. We manually check the code for such missing checks, and in certain cases, provide a proof that the given set of range checks are sufficient to constrain the circuit up to specification.

Cryptography. ZKP technology and their applications are based on various aspects of cryptography. We manually review the cryptography usage of the project and examine the relevant studies and standards for any inconsistencies or vulnerabilities.

Code maturity. We look for potential improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards.

For each finding, Zellic and KALOS assign it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

We organize its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

2.3 Scope

The engagement involved a review of the following targets:

Axiom Circuits

Repositories	https://github.com/axiom-crypto/axiom-eth-working
	https://github.com/axiom-crypto/axiom-eth-working/tree/develop/axiom-eth/src/mpt
	https://github.com/axiom-crypto/axiom-eth-working/pull/57
	https://github.com/axiom-crypto/axiom-eth-working/pull/33
	https://github.com/axiom-crypto/axiom-eth-working/tree/feat/sha-ssz-proof/axiom-eth/src/ssz
Versions	axiom-eth-working: `91a983d64407e16d24825be990155b701a722cdc`
	axiom-eth-working: `86b177157a05cb9c7e04a40c5132adf9decf0d0d`
	axiom-eth-working: `80cc96983eec2aa22ffefaeb91ff263439f22ea9`
	axiom-eth-working: `6fcbcfe782f081c04d1fe4676e1203bc04431cf4`

	axiom-eth-working: `e2406a0ef61781fd8e15d8302eb3af4b662c44e2`
Program	• axiom-eth/src/*
Types	Rust, Solidity
Platforms	Halo2, EVM

2.4 Project Overview

Zellic and KALOS were contracted to perform a security assessment with two consultants for a total of five person-weeks. The assessment was conducted over the course of three calendar weeks.

Contact Information

The following project manager was associated with the engagement:

Jasraj Bedi, Engagement Manager
jazzy@zellic.io

The following consultants were engaged to conduct the assessment:

Allen Roh, Engineer
allen@kalos.xyz

Mohit Sharma, Engineer
mohit@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

September 4, 2023	Kick-off
September 4, 2023	Start of primary review period
October 4, 2023	End of primary review period

3 Detailed Findings

3.1 The `verify_field_hash` function has incorrect Merkle proof-verification logic

- **Target**: src/ssz/mod.rs
- **Category**: Coding Mistakes
- **Likelihood**: High
- **Severity**: High
- **Impact**: High

Description

The `verify_field_hash` function, which aims to verify the value of a field at a certain position from an SSZ structure, takes an SSZ inclusion proof along with the maximum number of fields and the field index, then shows that the claimed value is included at the specified index.

The index can be proved by matching its bit representation with the direction values provided in the Merkle proof.

However, the given code matches the direction values with the byte representation of the index, instead of the bit representation. This is shown below.

```
pub fn verify_field_hash(
        &self,
        ctx: &mut Context<F>,
        field_num: AssignedValue<F>,
        max_fields: usize,
        proof: SSZInputAssigned<F>,
    ) -> SSZInclusionWitness<F> {
        assert!(max_fields > 0);
        let log_max_fields = log2(max_fields);
        self.range().check_less_than_safe(ctx, field_num, max_fields
    as u64);
        let field_num_bytes =
            uint_to_bytes_be(ctx, self.range(), &field_num,
    log_max_fields as usize); // byte representation
        let witness = self.verify_inclusion_proof(ctx, proof);
        let bad_depth = self.range().is_less_than_safe(ctx,
    witness.depth, log_max_fields as u64);
        self.gate().assert_is_const(ctx, &bad_depth, &F::from(0));
        for i in 1..(log_max_fields + 1) {
            let index = self.gate().sub(ctx, witness.depth,
    Constant(F::from(i as u64)));
            let dir_bit = self.gate().select_from_idx(ctx,
    witness.directions.clone(), index);
            ctx.constrain_equal(&dir_bit, field_num_bytes[(log_max_fields
    - i) as usize].as_ref());
        }
        witness
    }
```

Impact

As the directions are constrained to be boolean in the `verify_inclusion_proof`, any `field_num` value that has nonboolean value in the byte representation cannot be used in the `verify_field_hash` function. This implies that the circuit does not satisfy completeness.

Recommendations

We recommend using the bit representation to compare with the direction values.

Remediation

This issue has been acknowledged by Axiom, and a fix was implemented in commit 6e2f7454.

3.2 Insufficient maximum depth for the MPT proofs leads to a potential DOS attack

- **Target**: src/storage/mod.rs
- **Category**: Coding Mistakes
- **Likelihood**: High
- **Severity**: High
- **Impact**: High

Description

As seen in the src/storage/mod.rs code, the maximum depth for the account proof is set to 10. This value is sent over to the MPT circuits as the `max_depth` value.

```
pub const ACCOUNT_PROOF_MAX_DEPTH: usize = 10;
pub const STORAGE_PROOF_MAX_DEPTH: usize = 9;
```

However, given a target address, it is feasible to compute private keys corresponding to addresses that make the MPT inclusion proof for the target address have depth larger than 10. For example, simply working with the first 11 hex values, one can run a parallelizable `O(2^44)` attack to find the relevant private keys.

Impact

All storage proofs or account proofs relevant to the targeted address will fail, leading to a denial-of-service–like impact.

Recommendations

We recommend increasing the maximum depth of the account proofs and storage proofs accordingly.

Remediation

Axiom acknowledged this finding and provided the below response.

> 1. We have moved these constants to axiom-query in the second audit: axiom-query
>
> 2. In a subsequent PR, we added max_trie_depth to core_params for Account, Storage, Transaction, and Receipt subquery circuits, so they are accurately recorded as circuit configuration parameters.

In production, we will use the following max_trie_depth's:

- Account(state) trie: 14
- Storage trie: 13
- Transaction trie: 6
- Receipt trie: 6

For account and storage, these max depths were determined by running an analysis on a Geth full node: https://hackmd.io/@axiom/BJBledudT.

3.3 Function `new_from_bytes` in src/ssz/types.rs is incorrect

- **Target**: src/ssz/types.rs
- **Category**: Coding Mistakes
- **Likelihood**: N/A
- **Severity**: Informational
- **Impact**: Informational

Description

The `new_from_bytes` function in `SszBasicTypeList<F>` takes a vector of `AssignedBytes<F>` and the `len` to create a new `SszBasicTypeList<F>`. To do so, it computes the `pre_len` array, which represents whether or not the current index is less than the `len` value.

The point of computing this array, as shown in other functions such as `new_mask`, is that the value can be multiplied by the `pre_len` array to force all values at index no less than `len` to be equal to zero. This is shown in the code below.

```
pub fn new_mask(
        ctx: &mut Context<F>,
        range: &RangeChip<F>,
        values: Vec<SszBasicType<F>>,
        int_bit_size: usize,
        len: AssignedValue<F>,
) -> Self {
    // ...
    for j in 0..values.len() {
        let mut new_bytes = Vec::new();
        for i in 0..int_byte_size {
            let val = range.gate().mul(ctx, values[j].value()[i],
    pre_len[j]);
            new_bytes.push(val);
        }
        let new_basic = SszBasicType::new(ctx, range, new_bytes,
    int_bit_size);
        new_list.push(new_basic);
    }
    // ...
}
```

Here, we see that all bytes in the `values[j].value()` are multiplied with `pre_len[j]` correctly. However, in the `new_from_bytes` function, this is handled incorrectly.

```
pub fn new_from_bytes(
        ctx: &mut Context<F>,
        range: &RangeChip<F>,
        vals: Vec<AssignedBytes<F>>,
        int_bit_size: usize,
        len: AssignedValue<F>,
) -> Self {
    // ...
    for value in vals {
        let mut new_value = Vec::new();
        for i in 0..32 {
            let new_val = range.gate.mul(ctx, value[i], pre_len[i]);
            new_value.push(new_val);
        }
        let basic_type = SszBasicType::new(ctx, range, new_value,
    int_bit_size);
        values.push(basic_type);
    }
    // ...
}
```

Here, we see that `value[i]`, which is the `i`th byte of a single `AssignedBytes<F>` instance, is multiplied with the `pre_len[i]`, which is incorrect.

We also note that the `pre_len` array is initialized with the length `values.len()`, which is zero.

```
pub fn new_from_bytes(
        ctx: &mut Context<F>,
        range: &RangeChip<F>,
        vals: Vec<AssignedBytes<F>>,
        int_bit_size: usize,
        len: AssignedValue<F>,
) -> Self {
    // ...
    let mut values = Vec::new();
    // safety constraints?
    let len_minus_one = range.gate.dec(ctx, len);
    let len_minus_one_indicator = range.gate.idx_to_indicator(ctx,
    len_minus_one, vals.len());
    let zero = ctx.load_zero();
    let mut pre_len = vec![zero; values.len()];
    // ...
}
```

To the best of our knowledge, this function is not used anywhere.

Recommendations

We recommend removing the `new_from_bytes` function.

Remediation

This issue has been acknowledged by Axiom, and a fix was implemented in commit 54dabf29.

3.4 The node type of terminal node in MPT is not range checked to be a bit

- **Target**: src/mpt/mod.rs
- **Category**: Coding Mistakes
- **Likelihood**: Medium
- **Severity**: Medium
- **Impact**: Medium

Description

All inputs to the MPT inclusion/exclusion proof circuit are range checked in `parse_mpt_inclusion_phase0` to ensure there is no undefined behavior in functions that expect input witness values to be bytes or boolean values. The `node_type` for every node in proof is range checked to be a single bit; however, this check is missed for `proof.leaf.node_type`.

```
for bit in iter::once(&proof.slot_is_empty)
    .chain(proof.nodes.iter().map(|node| &node.node_type))
    .chain(proof.key_frag.iter().map(|frag| &frag.is_odd))
{
    self.gate().assert_bit(ctx, *bit);
}
```

Impact

This missing range check can lead to undefined behavior as `proof.leaf.node_type` is passed into functions that assume the corresponding argument to be boolean, such as in `parse_terminal_node_phase0`.

```
self.gate().select(ctx, node_byte, dummy_ext_byte, leaf_bytes.node_type)
self.gate().select(ctx, dummy_branch_byte, node_byte,
    leaf_bytes.node_type)
```

Recommendations

Assert `proof.leaf.node_type` to be boolean.

```
for bit in iter::once(&proof.slot_is_empty)
    .chain(proof.nodes.iter().map(|node| &node.node_type))
    .chain(proof.key_frag.iter().map(|frag| &frag.is_odd))
    .chain(vec![proof.leaf.node_type])
{
    self.gate().assert_bit(ctx, *bit);
}
```

Remediation

This issue has been acknowledged by Axiom, and a fix was implemented in commit 3ff70a54.

3.5 No leading zero check in rlp(idx) leads to soundness bug in transaction circuit

- **Target**: src/transaction/mod.rs
- **Category**: Coding Mistakes
- **Likelihood**: High
- **Severity**: Critical
- **Impact**: Critical

Description

The transaction trie maps `rlp(transaction_index)` to the `rlp(transaction)` or `TxType | rlp(transaction)`, depending on whether the transaction is a legacy transaction or not. One of the goals of the transaction circuit is to validate whether `transaction_index` exists in the trie or not.

To do so, the circuit validates that the `key_bytes` of the `MPTProof<F>` structure is equal to the RLP-encoded `transaction_index`. This is done as follows — first, the `key_bytes` is RLP decoded. Then, the decoded bytes are evaluated as an integer. Then, the evaluated value is constrained to be equal to the `transaction_index`.

```
pub fn parse_transaction_proof_phase0(
        &self,
        ctx: &mut Context<F>,
        input: EthTransactionInputAssigned<F>,
) -> EthTransactionWitness<F> {
    // ...
    // check key is rlp(idx):
    // given rlp(idx), parse idx as var len bytes
    let idx_witness = self.rlp().decompose_rlp_field_phase0(
        ctx,
        proof.key_bytes.clone(),
        TRANSACTION_IDX_MAX_LEN,
    );
    // evaluate idx to number
    let tx_idx =
        evaluate_byte_array(ctx, self.gate(), &idx_witness.field_cells,
    idx_witness.field_len);
    // check idx equals provided transaction_index from input
    ctx.constrain_equal(&tx_idx, &transaction_index);
    // ...
}
```

Here, the `TRANSACTION_IDX_MAX_LEN` is set to 2. This may cause an issue, as there is no check that the RLP-decoded bytes have no leading zeros. In the case where `transaction_index = 4`, the actual transaction is stored in the key `rlp(0x04)`. However, one can set the `key_bytes` as `rlp(0x0004)` and it would still satisfy all the constraints.

The issue is that there would not be any value corresponding to the key `rlp(0x0004)`, so even when there is actually a transaction with index 4, it would be possible to prove that there is no such a transaction.

A similar issue is also present in the receipt circuit.

Impact

This can be used create a fake proof that a block has an incorrect number of transactions. Suppose that there are actually 20 transactions in a block. One can prove that a transaction with index 4 exists in the block as usual, then prove that a transaction with index 5 does not exist in the block using the vulnerability we describe above. This is sufficient to prove that there are only five transactions in the block.

Recommendations

We recommend adding a padding check to the RLP decomposition.

Remediation

This issue has been acknowledged by Axiom, and a fix was implemented in commit f3b1130e.

3.6 Underconstrained circuit in length proofs for transaction circuit

- **Target**: src/transaction/mod.rs
- **Category**: Coding Mistakes
- **Likelihood**: High
- **Severity**: Critical
- **Impact**: Critical

Description

The transaction circuit (which we analyze further in section 4.2) aims to validate whether a transaction index exists in the transaction trie as well as what transaction corresponds to the index. It can also provide a proof for the number of transactions in a certain block. For this, the list of checks is as follows.

- Checks if the transaction root is `KECCAK_RLP_EMPTY_STRING` and sets `is_empty` to be true if this is the case
- Checks if either `noninclusion_idx - inclusion_idx = 1` or `noninclusion_idx = 0`
- Shows that either the inclusion proof (`len_proof[0]`) for the `inclusion_idx` is a proper inclusion proof or `is_empty` is true
- Shows that the noninclusion proof (`len_proof[1]`) for the `noninclusion_idx` is a proper noninclusion proof

For example, in the case where `inclusion_idx = 4` and `noninclusion_idx = 5`, one proves that index 4 exists yet index 5 does not, showing that there are exactly five transactions, corresponding to indexes 0, 1, 2, 3, and 4.

However, this list of checks is insufficient, as in the case where `is_empty` is true, one can set `inclusion_idx` as whatever value and set `noninclusion_idx` as `inclusion_idx + 1`. This is caused by missing the check that `is_empty` being true must force `noninclusion_idx = 0`.

Impact

One can prove that there are many transactions for a block that actually has no transactions.

Recommendations

We recommend adding the check that `is_empty` being true implies that `noninclusion_idx = 0`.

Remediation

This issue has been acknowledged by Axiom, and a fix was implemented in commit 6296e361.

4 Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment.

4.1 Analysis of the storage circuit

We summarize the constraints in the storage circuit.

The circuit has the block hash, block number, and address as well as the slot-value pairs as the public instances, and it checks that all MPT proofs (account proofs and storage proofs) are MPT inclusion proofs.

Looking into src/storage/mod.rs, we can summarize the constraints on each function as follows.

`parse_account_proof_phase0`

- hashes the address via `keccak_fixed_len` and constrains that it is equal to the `key_bytes` of the `MPTProof<F>`. It RLP decomposes the value retrieved from the state trie via `decompose_rlp_array_phase0`, then runs the MPT inclusion proof via `parse_mpt_inclusion_phase0`.

`parse_account_proof_phase1`

- finishes the RLP decomposition and the MPT inclusion proof via according phase1 functions and computes the `RlcTrace<F>` of each entry (nonce, balance, storage root, code hash).

`parse_storage_proof_phase0`

- hashes the slot via `keccak_fixed_len` and constrains that it is equal to the `key_bytes` of the `MPTProof<F>`. It RLP decomposes the value retrieved from the storage trie via `decompose_rlp_field_phase0`, then runs the MPT inclusion proof via `parse_mpt_inclusion_phase0`.

`parse_storage_proof_phase1`

- finishes the RLP decomposition and the MPT inclusion proof via according phase1 functions, then computes the `RlcTrace<F>` of the value.

`parse_eip1186_proofs_phase0`

- runs `parse_account_proof_phase0` and all `parse_storage_proof_phase0s` — and constrains that the `root_hash_bytes` for all storage proofs are equal to the `stora `ge_root` from the account trace.

`parse_eip1186_proofs_phase1`

- runs `parse_account_proof_phase1` and `parse_storage_proofs_phase1`.

`parse_eip1186_proofs_from_block_phase0`

- decomposes the block header with `decompose_block_header_phase0`;
- from the block header, fetches `state_root`, `block_hash`, and `block_number`;
- loads the address as well as the slots and the proofs;
- runs `parse_eip1186_proofs_phase0` ;
- constrains that the account proof's `root_hash_bytes` matches the `state_root`; and
- loads the slot-value pairs into hi-lo form and collects them.

`parse_eip1186_proofs_from_block_phase1`

- runs `decompose_block_header_phase1`, and
- runs `parse_eip1186_proofs_phase1`.

Overall, the circuits constrain that

- all MPT proofs are inclusion proofs,
- all RLP decompositions are done correctly,
- the key for the MPT proof is the hashed address / hashed slot,
- account proof's root is the state root from the block header, and
- storage proof's root is the storage root from the state trie.

4.2 Analysis of the transaction circuit

We summarize the constraints in the transaction circuit.

Looking into src/transaction/mod.rs, we can summarize the constraints on each function as follows.

`parse_transaction_proof_phase0`

- checks that the `key_bytes` of the MPT proof RLP decomposes into a byte array that evaluates to the transaction_index;

- checks that MPT inclusion proof is correct;
- computes the maximum field length based on the maximum data length, maximum access list length, and the enabled transaction types;
- asserts that if every type is disabled, the MPT proof is a noninclusion proof;
- checks whether type is nonzero by comparing the first value byte against 128. By multiplying type_is_not_zero with the first value byte, one can compute the tx_type accordingly;
- by replacing tx_type with tx_type * slot_is_full - slot_is_empty, forces tx_type = -1 when slot_is_empty is true (noninclusion proof)
- based on type_is_not_zero, selects the ith byte of the actual transaction;
- if slot_is_empty, replaces the transaction bytes with 0xc100 = rlp(0x00); and
- RLP decomposes the transaction bytes as an array.

parse_transaction_proof_phase1

- in phase1, finishes the RLP decomposition of the MPT key bytes;
- in phase1, finishes the MPT inclusion/exclusion proof; and
- in phase1, finishes the RLP decomposition of the transaction bytes.

parse_transaction_proofs_from_block_phase0

- decomposes the block header to get the transaction root;
- checks the transaction MPT proof's root_hash_bytes is the transaction root;
- checks if the transaction root is KECCAK_RLP_EMPTY_STRING and sets is_empty to be true if this is the case;
- checks if either noninclusion_idx - inclusion_idx = 1 or noninclusion_idx = 0;
- shows that the inclusion proof (len_proof[0]) is a proper inclusion proof or is_empty is true (this is weak, as shown in Finding);
- shows that the noninclusion proof (len_proof[1]) is a noninclusion proof; and
- checks that all length proofs' transaction proof is correct and that their MPT

proof's `root_hash_bytes` is the transaction root.

`parse_transaction_proofs_from_block_phase1`

- runs block header decomposition in phase1, and
- runs the transaction proof parsing in phase1.

`parse_transaction_proof_from_block_phase0`

- runs block header decomposition in phase0 to get the transaction root,
- parses the transaction proof in phase0, and
- checks that the transaction root is the MPT proof's `root_hash_bytes`.

`parse_transaction_proof_from_block_phase1`

- runs block header decomposition in phase1, and
- runs the transaction proof parsing in phase1.

`extract_field`

- checks that the proof is an inclusion proof, and
- selects each field byte and the length with an indicator.

4.3 Analysis of the receipt circuit

We summarize the constraints in the receipt circuit.

Looking into src/receipt/mod.rs, we can summarize the constraints on each function as follows.

`parse_receipt_proof_phase0`

- checks that the `key_bytes` of the MPT proof RLP decomposes into a byte array that evaluates to the `transaction_index`;
- checks that the MPT inclusion proof is correct;
- checks whether type is nonzero by comparing the first value byte against 128. By multiplying `type_is_not_zero` with the first value byte, one can compute the `tx_type` accordingly;
- by replacing `tx_type` with `tx_type * slot_is_full - slot_is_empty`, forces `tx_type = -1` when `slot_is_empty` is true (noninclusion proof);

- based on `type_is_not_zero`, selects the `i`th byte of the actual receipt;
- if `slot_is_empty`, replaces the receipt bytes with `0xc100 = rlp(0x00)`; and
- RLP decomposes the receipt bytes, then RLP decomposes the log array.

`parse_receipt_proof_phase1`

- in phase1, finishes the RLP decomposition of the MPT key bytes;
- in phase1, finishes the MPT inclusion/exclusion proof; and
- in phase1, finishes the RLP decomposition of the receipt bytes and the log array.

`extract_receipt_field`

- selects each field byte and the length with an indicator.

`extract_receipt_log`

- selects each log byte and the length with an indicator.

4.4 Analysis of the Solidity circuit

We summarize the constraints in the Solidity circuit.

Looking into src/solidity/mod.rs, we can summarize the constraints on each function as follows.

`slot_for_mapping_value_key`

- concats the `key` and the `mapping_slot` (both `SafeBytes32<F>`) and applies `keccak_fixed_len`.

`slot_for_mapping_nonvalue_key_phase0`

- concats the `key` and the `mapping_slot` directly without constraining it and applies `keccak_var_len`.

`slot_for_mapping_nonvalue_key_phase1`

- proves that the keccak hashed byte array is the `key` with the `mapping_slot` by standard RLC techniques.

`slot_for_mapping_key_phase0`

- depending on the type of the `key`, delegates the logic to `slot_for_mapping_value_key` or `slot_for_mapping_nonvalue_key_phase0`.

`slot_for_mapping_key_phase1`

- depending on the type of the `key`, delegates the logic to nothing or `slot_for_mapping_nonvalue_key_phase1`.

`slot_for_nested_mapping_phase0`

- iteratively computes the nested slot with `slot_for_mapping_key_phase0`, and
- with an indicator, selects the appropriate nested slot.

`slot_for_nested_mapping_phase1`

- runs `slot_for_mapping_key_phase1` for each nested slot.

`verify_mapping_storage_phase0`

- computes the nested slot with `slot_for_nested_mapping_phase0`, and
- parses the storage proof with `parse_storage_proof_phase0`.

`verify_mapping_storage_phase1`

- runs `slot_for_nested_mapping_phase1`, and
- finishes the storage proof with `parse_storage_proof_phase1`.

4.5 Analysis of the block header circuit

We summarize the constraints in the block header circuit.

The circuit has

- the previous block hash,
- the end block hash,
- block numbers (start/end numbers concat), and
- Merkle mountain range peaks

as the public instance. The circuit checks that

- block headers are decomposed appropriately,
- `num_blocks ≤ 2^max_depth`,
- the boundary data (from the public instances) are computed correctly, and

- the Merkle mountain range is computed correctly according to the number of blocks.

Looking into src/block_header/mod.rs, we can summarize the constraints on each function as follows.

`get_number_fixed`

- left pads the `number` to four bytes into `FixLenBytes<F, 4>`.

`get_number_value`

- evaluates the `number` byte array into an `AssignedValue<F>`.

`decompose_block_header_phase0`

- range checks that all block header elements are bytes,
- RLP decomposes the block header in phase0, and
- runs `keccak_var_len` on the RLP array to compute the block hash.

`decompose_block_header_phase1`

- RLP decomposes the block header in phase1, and
- computes the `RlpFieldTrace<F>` for each block header entry.

`decompose_block_header_chain_phase0`

- parallelizes instances of `decompose_block_header_phase0`.

`decompose_block_header_chain_phase1`

- runs parallel instances of `decompose_block_header_phase1`, and
- checks that the block headers form a chain (i.e., that the next block's `parent_hash` matches the current block's block hash).

`get_boundary_block_data`

- fetches the `prev_block_hash` by hi-lo decomposition of the first `parent_hash_bytes`,
- fetches `end_block_hash` by selecting `block_hash.hi_lo()`s with the indicator,
- fetches `start_block_number_bytes` via `get_number_fixed` on the first block,
- fetches `end_block_number_bytes` by selecting `get_number().field_cells` and the `get_number().field_len` with the indicator,

- computes the `block_numbers` by concatenating the two fixed bytearrays and evaluating into a single `AssignedValue<F>`,

4.6 Analysis of the SSZ circuit

We summarize the constraints in the SSZ circuit.

Looking into src/ssz/mod.rs and src/ssz/types.rs, we can summarize the constraints on each function as follows.

`SszBasicType<F>`

`new`

- range checks each value according to the `int_bit_size`.

`new_from_unassigned_vec`

- loads raw bytes as witnesses and calls `new`.

`new_from_int`

- computes the little endian byte representation and calls `new_from_unassigned_vec`.

`SszBasicTypeVector<F>`

`new`

- asserts that each value's `int_bit_size` is consistent.

`new_from_bytes`

- calls `SszBasicType::new` and pushes each value.

`new_from_unassigned_vecs`

- calls `SszBasicType::new_from_unassigned_vecs` and pushes each value .

`new_from_ints`

- calls `SszBasicType::new_from_ints` and pushes each value.

`SszBasicTypeList<F>`

`new`

- checks that `len ≤ values.len()`,

- asserts that `int_bit_sizes` are consistent, and
- checks that `values[len..]` is all zeros; this is done by maintaining a suffix sum of an indicator and asserting that if the suffix sum is zero, then the value must be zero as well.

`new_mask`

- checks that `len ≤ values.len()`,
- asserts that `int_bit_sizes` are consistent, and
- multiplies the previously mentioned suffix sum to the value directly, forcing the value to be zero when the suffix sum is zero.

`new_from_unassigned_vecs`

- simply asserts without constraining that `len ≤ vals.len()`,
- simply asserts without constraining that `vals[len..]` is all zeros, and
- is not used in actual circuits.

`new_from_ints`

- simply asserts without constraining that `len ≤ vals.len()`,
- simply asserts without constraining that `vals[len..]` is all zeroes, and
- is only used in the test circuits.

Inclusion proofs

`verify_inclusion_proof`

- checks `depth ≠ 0` and `depth ≤ max_depth`,
- checks `directions` are boolean,
- verifies the Merkle proof with the direction values, and
- checks that the final Merkle root is equal to the `root_bytes`.

`verify_field_hash`

- checks `field_num < max_fields`,
- checks `depth ≥ log_max_fields`, and

- matches the bit representation of `field_num` with the direction bits.

`verify_struct_inclusion`

- runs `verify_inclusion_proof`, and
- checks that the struct's `hash_root` is equal to the witness value.

`verify_struct_field_inclusion`

- runs `verify_field_hash`, and
- checks that the struct's `hash_root` is equal to the witness value.

4.7 Analysis of the MPT circuit

The MPT circuit is responsible for MPT inclusion and exclusion proofs. We summarize the constraints in the MPT circuit.

Looking into src/mpt/mod.rs, we can summarize the constraints on each function as follows.

`parse_mpt_inclusion_phase0`

- range checks all inputs,
- parse nodes RLP using `decompose_rlp_array_phase0`, and
- checks key fragment and prefix consistency.

`parse_mpt_inclusion_phase1`

- checks RLP consistency via RLC concatenation;
- constrains that for all extension nodes, the key fragment corresponds to the corresponding fragment in `key_frags`. If `slot_is_empty` is true, the same check is performed for all nodes except the terminal node. In this case, it is constrained that if the terminal node is extension, the node's fragment must not equal the last key fragment in `key_frags`;
- constrains that key fragments concatenate to key;
- constrains that `value_bytes` matches the value read from the leaf node; and
- constrains the Merkle hash chain.

4.8 Analysis of the RLP circuit

We summarize the constraints in the RLP circuit.

Looking into src/rlp/mod.rs, we can summarize the constraints on each function as follows.

`decompose_rlp_field_phase0`

- range checks the prefix byte,
- constrains `len_len` to the prefix, and
- witnesses the field cells (without constraining).

`decompose_rlp_array_phase0`

- range checks the prefix byte;
- constrains `len_len` to the prefix;
- assumes a fixed number of elements and iterates through;
- if `is_variable_len` is true, constrains the prefix of each element in the array to be less than the len parsed from prefix; and
- witnesses the field cells for all elements in the array (without constraining).

`decompose_rlp_field_phase1`

- computes RLC of the length cells and field cells, and
- constrains the concatenation of len cells and field cells to be equal to the field RLP.

`decompose_rlp_array_phase1`

- computes the RLC of the length cells and field cells, and
- constrains the concatenation of the RLP of all field cells to be equal to the RLP of the complete array.

4.9 Analysis of the RLC circuit

We summarize the constraints in the RLC circuit.

Looking into src/rlc/mod.rs, we can summarize the constraints on each function as follows.

`compute_rlc_with_min_len`

- Given an input vector (`val_1`, `val_2`, `val_3` ... `val_n`), it computes and assigns witnesses to the RLC values.
- The `ctx_rlc.advice` looks like `| rlc0=val0 | val1 | rlc1 | val2 | rlc2 | ... | rlc_{max_len - 1} |`.
- Final RLC index is constrained to be zero if the input length is zero, length - 1 otherwise.
- It indexes into the calculated RLC advice row using the final RLC index to compute RLC.

4.10 Analysis of the keccak circuit

Axiom uses a standalone keccak coprocessor circuit for keccak computations. The coprocessor circuit was not part of this audit's scope, but the keccak module defines the `KeccakManager` chip, which is essentially used to communicate with and manage the coprocessor circuit.

Essentially, the keccak module performs the following functions:

- It computes and assigns witnesses (without constraining) for keccak outputs.
- The input/output pairs used are passed to the `KeccakManager`.
- In the second phase, `KeccakManager` constructs a dynamic lookup table of all requests it received. Each row in the lookup table represents the 3-tuple `(rlc_encoding(input_assigned), output_hi, output_lo)`.
- The `KeccakManager` performs a lookup for all input/output pairs and outputs a commitment to the table as a public instance.
- The validity of the table is ensured by comparing the table commitment generated by `KeccakManager` with that output by the keccak coprocessor circuit.

`KeccakChip<F>`

`keccak_var_len`

- range checks `len`,
- inserts the corresponding request into `manager.state`,

- loads the hi-lo bytes of the output **unconstrained** (to be constrained through lookup by the manager in `SecondPhase`), and
- inserts the keccak query to `manager.var_len_queries`.

`keccak_fixed_len`

- inserts the corresponding request into `manager.state`,
- loads the hi-lo bytes of the output **unconstrained** (to be constrained through lookup by the manager in `SecondPhase`), and
- inserts the keccak query to `manager.var_len_queries`.

`KeccakManager<F>`

`generate_witnesses_phase0_end`

- is called at the end of the first phase — `manager.state` must be `AcceptingRequests`;
- initializes Poseidon spec (expensive but only done once in the lifetime of a circuit);
- encodes every request in `manager.state` (constrained) amd outputs a commitment to be matched with the coprocessor output;
- takes the resulting output commitment and assigns it as an instance; and
- transitions `manager.state` to `CoprocessorEncoded`.

`raw_synthesize_phase0_end`

- is called at the end of the first phase — `manager.state` must be `CoprocessorEncoded`, and
- assigns `encoding.loaded_keccak_fs.hash_hi` and `encoding.loaded_keccak_fs.hash_lo` to hash output columns of the dynamic lookup table.

`generate_witnesses_phase1_start`

- is called at the beginning of SecondPhase witness generation — `manager.state` must be `CoprocessorEncoded`, and
- transitions `manager.state` to `Finalized`.

5 Audit Results

At the time of our audit, the audited code was not deployed to mainnet.

During our assessment on the scoped Axiom circuits, we discovered six findings. Two critical issues were found. Two were of high impact, one was of medium impact, and the remaining finding was informational in nature. Axiom acknowledged all findings and implemented fixes.

5.1 Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic and KALOS, of course, also cannot make guarantees about any code added to the project after the audit version of our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, we provide a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic or KALOS.