Axiom

Security Assessment

November 10, 2023

Prepared for:

Yi Sun, Jonathan Wang

Axiom

Prepared by:

Malte Leip, Gyumin 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 October 30th to November 10th, 2023. During this engagement, we reviewed Axiom's code for security vulnerabilities, design issues, and general weaknesses in security posture.

Following the completion of this audit, Axiom requested our assessment of three pull requests:

- PR 213 Remove `header_max_field_bytes` from CoreParamsHeaderSubquery
- PR 218 Remove hard-coded block header constants
- PR 219 Update for Cancun

No security issues were identified in association with these particular updates.

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.

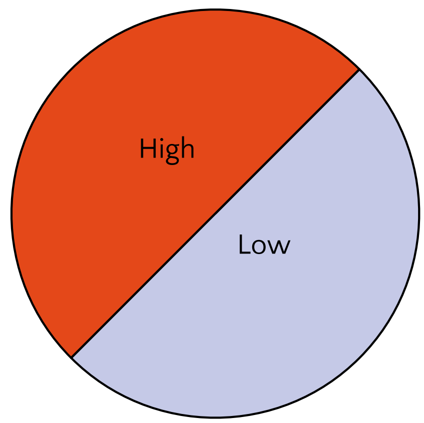
1.3 Results

During our assessment on the scoped Axiom circuits, we discovered two findings. No critical issues were found. One finding was of high impact and one was of low impact.

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

Breakdown of Finding Impacts

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



2 Introduction

2.1 About Axiom

Axiom scales data-rich applications on Ethereum by providing smart contracts trustless access to historic on-chain data and verified compute over it.

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

Repository	https://github.com/axiom-crypto/axiom-eth-working
Version	axiom-eth-working: `7c0bf7b10e401f8a1a7df44ff9272f58b257ffac`
Program	• axiom-query/src/components/subqueries/*
Types	Rust, Solidity
Platforms	Halo2, EVM

2.4 Project Overview

Zellic and KALOS were contracted to perform a security assessment with three 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:

Chad McDonald, Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Malte Leip, Engineer
malte@zellic.io

Gyumin Roh, Engineer
rkm0959@kalos.xyz

Mohit Sharma, Engineer
mohit@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

October 30, 2023	Start of primary review period
November 10, 2023	End of primary review period

3 Detailed Findings

3.1 The `decompose_rlp_array_phase1` is missing in receipt-query circuits

- **Target**: receipt/circuit.rs
- **Category**: Coding Mistakes
- **Likelihood**: High
- **Severity**: High
- **Impact**: High

Description

The receipt circuit deals with the receipts and the parsing of receipts into various fields and logs as well as the parsing of logs into topics and data. One of the main functions inside the receipt-query circuit is the `parse_log` function, which parses a log by decomposing the RLP encoded byte array into a list of addresses, topics, and data. The topics byte array is then once again RLP decoded into a list of topics. These two RLP decompositions are done via the `RlpChip`'s `decompose_rlp_array_phase0`.

However, unlike every other usage of `decompose_rlp_array_phase0`, there is no corresponding `decompose_rlp_array_phase1` being done on the `RlpArrayWitness<F>` at the relevant phase. This leads to a soundness issue.

Impact

The RLP decomposition of the logs into addresses, topics, and data and the RLP decomposition of topics into a variable length list of topics is underconstrained.

Recommendations

We recommend adding the `decompose_rlp_array_phase1` calls appropriately to avoid soundness vulnerabilities.

Remediation

This issue has been acknowledged by Axiom, and fixes were implemented in the following commits:

- 4f73b7bb
- 5985b263

3.2 Contract data being short leads to completeness bug in the transaction query circuit

- **Target**: transaction/circuit.rs
- **Category**: Coding Mistakes
- **Likelihood**: Low
- **Severity**: Medium
- **Impact**: Low

Description

The transaction-query circuit aims to answer various queries about a transaction in the EVM. One of the possible queries is about the `data` of a transaction; it can be asked in two ways. The first is as contract data, and the second is as calldata. In the case of contract data, the entire `data` is considered, but in the case of calldata, the first four bytes are omitted as they are considered to be the function selector. If a calldata has less than four bytes, then it is considered to be an invalid calldata.

To implement this, first the `buffer` is prepared as the data bytes with the first four bytes omitted if it is a calldata query and the raw data bytes if it is a contract-data query. This is done with a select gate as follows.

```
let buffer = (0..data_bytes.len())
    .map(|i| {
        if i + 4 < data_bytes.len() {
            gate.select(ctx, data_bytes[i + 4], data_bytes[i],
    in_calldata_range) // if calldata, take i + 4, else take i
        } else {
            gate.mul_not(ctx, in_calldata_range, data_bytes[i]) // if
    calldata, take \x00, else take i
        }
    })
    .collect_vec();
```

Then, the actual buffer length is computed as follows.

- `is_valid_calldata = (data_len ≥ 4)`
- `buffer_len = is_valid_calldata * (data_len - 4 * is_calldata_query)`

```
let is_valid_calldata =
    is_gte_usize(ctx, range, data_len, 4, bit_length(data_bytes.len()
    as u64));
```

```
let mut buffer_len = gate.sub_mul(ctx, data_len, Constant(F::from(4)),
    in_calldata_range);
buffer_len = gate.mul(ctx, buffer_len, is_valid_calldata);
let is_in_range =
    gate.mul_add(ctx, in_calldata_range, is_valid_calldata,
    in_contract_data_range);
shift = gate.mul(ctx, shift, is_in_range);
let (buffer, is_lt_len)
    = extract_array_chunk_and_constrain_trailing_zeros(
    ctx,
    range,
    &buffer,
    buffer_len,
    shift,
    32,
    FIELD_IDX_BITS,
);
```

This poses a problem. In the case where contract data is being queried and the data length itself is less than four, then `is_valid_calldata` will be false and `buffer_len` would be set to zero. However, the buffer does have nonzero bytes and the intended behavior would be to indeed return the contract-data bytes as normal.

However, as the buffer is considered to have zero length, the circuit will not behave as intended, leading to a completeness issue.

Impact

The contract data, in the case where its length is less than four, cannot be proved by the transaction query. This is a completeness issue. However, as it deals with a small portion of actual smart contracts, we marked this as a low-impact bug.

Recommendations

We recommend computing `buffer_len` appropriately to handle all cases. It should especially return `data_len` when it is a contract-data query regardless of whether or not it is less than four or not.

Remediation

This issue has been acknowledged by Axiom, and fixes were implemented in the following commits:

- 17e60b5e
- ab73ae2c

4 Discussion

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

4.1 RLP length bytes constraints

During exploration beyond the designated audit scope, we identified an issue in the RLP circuit. We have included these vulnerabilities for the sake of completeness.

The RLP decomposition circuit had two critical vulnerabilities in the encoding for strings and arrays, caused by underconstraints in length parsing. Both bugs basically had the same impact, allowing for multiple valid representations of the same string, which can in turn potentially be used to spoof exclusion proofs.

Below is an explanation and PoC of the two bugs:

1. In the RLP circuit, for long lists and strings (i.e `len_len` is non zero) the length is not constrained to be greater than 55. Therefore short strings with length less than 55 can be encoded using both the long and short convention, leading to multiple valid representations of the same list

Proof of concept:

```
pub fn attack() {
    let k = DEGREE;
    // the list [ "cat", "dog" ] = [ 0xc8, 0x83, 'c', 'a', 't', 0x83, 'd',
    'o', 'g' ]
    let cat_dog: Vec<u8> = vec![0xc8, 0x83, b'c', b'a', b't', 0x83, b'd',
    b'o', b'g'];
    let attack: Vec<u8> = vec![0xf8, 0x08, 0x83, b'c', b'a', b't', 0x83,
    b'd', b'o', b'g'];
    for mut test_input in [cat_dog, attack] {
        test_input.append(&mut vec![0; 69 - test_input.len()]);
        let circuit = rlp_list_circuit::<Fr>(
            CircuitBuilderStage::Mock,
            test_input,
            &[15, 9, 11, 10, 17],
            true,
            None,
```

```
            None,
        );
        MockProver::run(k, &circuit, vec![]).unwrap().assert_satisfied();
    }
}
```

2. When length is parsed for long strings, it was done by reading the number of bytes specified in `len_len` and then evaluating them to an integer value. There was a missing check for leading null bytes which allows an attack to add a number of `0x00` padding bytes to the length, again leading to multiple valid representations of the same list

Proof of concept:

```
pub fn attack() {
    let k = DEGREE;
    // the list [ "cat", "dog" ] = [ 0xc8, 0x83, 'c', 'a', 't', 0x83, 'd',
    'o', 'g' ]
    let cat_dog: Vec<u8> = vec![0xc8, 0x83, b'c', b'a', b't', 0x83, b'd',
    b'o', b'g'];
    let attack: Vec<u8> = vec![0xf9, 0x00, 0x08, 0x83, b'c', b'a', b't',
    0x83, b'd', b'o', b'g'];
    for mut test_input in [cat_dog, attack] {
        test_input.append(&mut vec![0; 300 - test_input.len()]);
        let circuit = rlp_list_circuit::<Fr>(
            CircuitBuilderStage::Mock,
            test_input,
            &[15, 9, 11, 10, 17],
            true,
            None,
            None,
        );
        MockProver::run(k, &circuit, vec![]).unwrap().assert_satisfied();
    }
}
```

This issue has been acknowledged by Axiom, and a fix was implemented in commit 37409288.

4.2 Analysis of the account-subquery circuit

We summarize the constraints in the account-subquery circuit.

The circuit has its output commitment as well as a promise commitment for calls to the header subquery as the public instances. The output commitment commits to a virtual table of key-value pairs `((block_number, addr, field_idx), value)`. Here, `block_number`, `addr`, and `field_idx` are field elements representing a block number, account address, and index in the Ethereum blockchain account-state tuple (which has four components, `nonce`, `balance`, `storageRoot`, and `codeHash`, in that order), respectively. Furthermore, `value` consists of two field elements that jointly encode 32 bytes of data (as a HiLo).

For each key, the circuit constrains `addr` to be a 160-bit value and $0 \leq$ `field_idx` < 4. For each value, the circuit introduces a witness `state_root`, a 256-bit value witnessed in two field elements as a HiLo. The witness `value` is constrained to be the HiLo representation of the 32 bytes obtained by padding with zero on the left of the `field_idx`-th component of the account-state tuple for the account with address `addr` at block number `block_number`, under the assumption that `state_root` is the Keccak hash of the root node of the state trie in block number `block_number`. If the address does not exist in the state trie, the `value` is constrained to a default value depending on `field_idx`. Verification of `state_root` is handled via a promise call to the header-subquery circuit.

Looking into src/components/subqueries/account/circuit.rs, we can summarize the constraints introduced by each function as follows.

`handle_single_account_subquery_phase0`

- Loads a witness `addr` for the address[1] and checks that it decomposes into 20 bytes, with `address` witnessing the decomposition.
- Assigns witnesses for an `MPTProof<F>` called `mpt_proof`.
- Uses `EthStorageChip::parse_account_proof_phase0` in combination with a later call to `EthStorageChip::parse_account_proof_phase1` in the second phase to verify that the MPT inclusion/exclusion proof `mpt_proof` is correct. Concretely, with the root of trust being the MPT root hash (the assumption is that it is the valid `stateRoot` for the block under consideration), this should ensure
  - that `keccak256(address)` is not included in the MPT trie iff `account_witness.mpt_witness().slot_is_empty` is true.

[1] Interpreted as a big endian number.

- that if `keccak256(address)` is included in the MPT trie, then the value is RLP decomposed into an `RlpArrayWitness<F>` with four components of lengths `[8, 12, 32, 32]`, with the result stored in `account_witness.array_witness()`.

- Constrains `state_root` to be the HiLo representation of the MPT root hash of `mpt_proof`. Thus, `state_root` can be considered the root of trust for `account_witness.mpt_witness().slot_is_empty` and `account_witness.array_witness()` now.
- Assigns a witness for `field_idx` and constrains it to satisfy $0 \leq$ `field_idx` < 4.
- Pads the components of `account_witness.array_witness()` on the left with zero bytes to normalize them to a length of 32 bytes and converts them to HiLo, storing the result in `account_fixed`.
- Replaces `account_fixed` by the default values to be used for nonexisting accounts if `account_witness.mpt_witness().slot_is_empty` is true.
- Extracts the `field_idx`-th component from `account_witness` and assigns it to `value`.
- Loads a witness for `block_number`.

Overall, this function returns

```
PayloadAccountSubquery {
    account_witness,
    state_root,
    output: AssignedAccountSubqueryResult {
        subquery: AssignedAccountSubquery { block_number, addr, field_idx
    },
        value,
    },
}
```

with the components constrained so that under the assumptions that

- the second phase constraints for `account_witness` hold, and
- the witness `state_root` is the correct `stateRoot` for the block with number `block_number`,

it holds that

- the witness `addr` is a 160-bit value,

- $0 \leq$ `field_idx` < 4, and
- the HiLo `value` contains the `field_idx`-th component of the account state of address `addr` at the block with number `block_number` if that account exists and a default value otherwise. The conversion to HiLo is as mentioned earlier.

`virtual_assign_phase0`

- Calls `handle_single_account_subquery_phase0` for each subquery (of its input shard).
- Computes the output commit for the virtual table of key-value pairs `((block_number, addr, field_idx), value)`, where these four components are those returned from `handle_single_account_subquery_phase0`.
- Makes a promise call, for each subquery, to the header-subquery component to obtain the `stateRoot` at the block with block number `block_number`, and constrains the result to be equal to the `state_root` witness contained in the return value from `handle_single_account_subquery_phase0`.

`handle_single_account_subquery_phase1`

- Calls `EthStorageChip::parse_account_proof_phase1` on the payloads's `account_witness`, thus verifying the remaining second-phase constraints.

`virtual_assign_phase1`

- Calls `handle_single_account_subquery_phase1` for each payload.

4.3 Analysis of the storage-subquery circuit

We summarize the constraints in the storage-subquery circuit.

The circuit has its output commitment as well as a promise commitment for calls to the account subquery as the public instances. The output commitment commits to a virtual table of key-value pairs `((block_number, addr, slot), value)`. Here, `block_number` and `addr` are field elements representing a block number and account address, and `slot` consists of two field elements that jointly encode the 32-byte key of a storage slot as a HiLo. Furthermore, `value` consists of two field elements that jointly encode 32 bytes of data as a HiLo.

For each key, the circuit constrains `slot` to be the HiLo representation of 32 bytes. For each value, the circuit introduces a witness `storage_root`, a 256-bit value witnessed in two field elements as a HiLo. The witness `value` is constrained to be the HiLo representation of the 32 bytes stored at storage slot `slot` of the account with address `addr` at the block with number `block_number`, under the assumption that `storage_root` is the Keccak hash of the root node of the storage trie of the account with address `addr` at block with number `block_number`. This assumption is verified via a promise call to the account-header subquery.

Looking into src/components/subqueries/storage/circuit.rs, we can summarize the constraints introduced by each function as follows.

`handle_single_storage_subquery_phase0`

- Loads a witness `addr` for the address.
- Loads 32 witnesses for the bytes of the slot as `slot_bytes` and constrains two field elements `slot` to be the HiLo representation of `slot_bytes`.
- Assigns witnesses for an `MPTProof<F>` called `mpt_proof`.
- Uses `EthStorageChip::parse_storage_proof_phase0` in combination with a later call to `EthStorageChip::parse_storage_proof_phase1` in the second phase to verify that the MPT inclusion/exclusion proof `mpt_proof` is correct. Concretely, with the root of trust being the MPT root hash (the assumption is that it is the valid `storageRoot` for the block under consideration), this should ensure
  - that `slot_bytes` indeed consists of bytes.
  - that `keccak256(slot_bytes)` is not included in the MPT trie iff `storage_witness.mpt_witness().slot_is_empty` is true.
  - that if `keccak256(slot_bytes)` is included in the MPT trie, then the value is RLP decoded into an `RlpFieldWitness<F>` of at most 32 bytes, with the result stored in `storage_witness.value_witness()`.
- Constrains `storage_root` to be the HiLo representation of the MPT root hash of `mpt_proof`. Thus, `storage_root` can be considered the root of trust for `storage_witness.mpt_witness().slot_is_empty` and `storage_witness.value_witness()` now.
- Constrains `value` to be the HiLo representation of `storage_witness.value_witness()` after padding on the left with zero bytes to a length of 32 bytes.
- Replaces `value` by the default value zero if `storage_witness.mpt_witness().slot_is_empty` is true.
- Loads a witness for `block_number`.

Overall, this function returns

```
PayloadStorageSubquery {
    storage_witness,
    storage_root,
    output: AssignedStorageSubqueryResult {
        subquery: AssignedStorageSubquery { block_number, addr, slot },
        value,
    },
}
```

with the components constrained so that under the assumptions that

- the second phase constraints for `storage_witness` hold, and
- the witness `storage_root` is the correct `storageRoot` for the account with address `addr` at the block with number `block_number`,

it holds that

- the two field elements `slot` are the HiLo representation of 32 bytes, and
- the HiLo `value` contains the value stored at storage key `slot` of the storage associated to the account with address `addr` at the block with number `block_number`.

`virtual_assign_phase0`

- Calls `handle_single_storage_subquery_phase0` for each subquery (of its input shard).
- Computes the output commit for the virtual table of key-value pairs `((block_number, addr, slot), value)`, where these four components are those returned from `handle_single_storage_subquery_phase0`.
- Makes a promise call, for each subquery, to the account-subquery component to obtain the `storageRoot` of the account with address `addr` at the block with block number `block_number`, and it constrains the result to be equal to the `storage_root` witness contained in the return value from `handle_single_storage_subquery_phase0`. The promise call will also check that `addr` is well-formed.

`handle_single_storage_subquery_phase1`

- Calls `EthStorageChip::parse_storage_proof_phase1` on the payloads's `storage_witness`, thus verifying the remaining second-phase constraints.

`virtual_assign_phase1`

- Calls `handle_single_storage_subquery_phase1` for each payload.

4.4 Analysis of the Solidity-mapping subquery circuit

We summarize the constraints in the Solidity-mapping subquery circuit.

The circuit has its output commitment as well as a promise commitment for calls to the storage subquery as the public instances. The output commitment commits to a virtual table of key-value pairs `((block_number, addr, mapping_slot, mapping_depth, keys), value)`. Here, `block_number` and `addr` are field elements representing a block number and account address, and `mapping_slot` is a HiLo instance encoding 32 bytes of the slot for the mapping. The `mapping_depth` (field elements) and `keys` (HiLo instances) represent the keys applied to the mapping. Furthermore, `value` consists of two field elements that jointly encode 32 bytes of data as a HiLo.

The circuit aims to compute the corresponding slot for the mapping's value based on `mapping_slot, mapping_depth, keys` and denotes this as `value_slot`. Then, a promise call to the storage subquery based on `(block_number, addr, value_slot)` is used to fetch the actual value on the `value_slot` slot as a HiLo instance. This is the `value` that is committed. A maximum of four Solidity-mapping keys are supported.

Looking into src/components/subqueries/solidity_mappings/circuit.rs, we can summarize the constraints introduced by each function as follows.

`handle_single_solidity_nested_mapping_subquery_phase0`

- Loads all 32 bytes of the `mapping_slot` and range checks them to be bytes. Each of the HiLos are checked to be 16 bytes by using `uint_to_bytes_be` on each of them.
- Loads all 32 bytes of all the `keys` and range checks them to be bytes. Similar to the `mapping_slot`, each of the HiLos are checked to be 16 bytes.
- Loads `mapping_depth, block_number, addr` as a witness.
- Computes the `mapping_witness` via the `slot_for_nested_mapping_phase0` function call.
- Converts the computed slot to a HiLo form to get `value_slot`.

Overall, this function returns

```
PayloadSolidityNestedMappingSubquery { mapping_witness, subquery,
    value_slot }
```

with the components constrained so that under the assumption that the second-phase constraints for `mapping_witness` hold, it holds that the witness `value_slot` is the HiLo form of the correctly computed slot.

`virtual_assign_phase0`

- Calls `handle_single_solidity_nested_mapping_subquery_phase0` for each subquery (of its input shard).
- Computes the output commit for the virtual table of key-value pairs `((block_number, addr, mapping_slot, mapping_depth, keys), value)`, where these components are those returned from `handle_single_solidity_nested_mapping_subquery_phase0`.
- Makes a promise call, for each subquery, to the storage-subquery component to obtain the storage value for the account with address `addr` at the block with block number `block_number` at the slot `value_slot` and fetches the result. The promise call will also check that `addr` is well-formed.

`handle_single_solidity_nested_mapping_subquery_phase1`

- Calls `SolidityChip::slot_for_nested_mapping_phase1` on the payload's `mapping_witness`, thus verifying the remaining second-phase constraints.

`virtual_assign_phase1`

- Calls `handle_single_solidity_nested_mapping_subquery_phase1` for each payload.

4.5 Analysis of the block-header–subquery circuit

We summarize the constraints in the block-header circuit.

The circuit has an output commitment that commits to a virtual table of key-value pairs `((block_number, field_idx), value)`. Here, `block_number` is the block number and `field_idx` is a field index that is being queried. Furthermore, `value` consists of two field elements that jointly encode 32 bytes of data as a HiLo.

Looking into src/components/subqueries/block_header/circuit.rs, we can summarize the constraints introduced by each function as follows.

`handle_single_header_subquery_phase0`

- Parses the RLP array for the block header (this automatically constrains the block number and block hash).
- Loads the MMR proof and verifies it with logic in mmr_verify.rs.
- Range checks the `field_idx` to satisfy `field_idx` $< 2^{32}$.
- Defines `is_idx_in_header = (field_idx < 50)`.
- Computes `header_idx = field_idx * is_idx_in_header`, so that it is `field_idx` if it is intended as a header index and zero otherwise.
- Pads the field witnesses accordingly based on whether a field is variable length or a field is of value type. If a field is variable length and value type, it suffices to left pad it into a fixed-length byte array.
- Truncates it into 32 bytes if the fixed byte array is longer than 32 bytes.
- Takes `len` to be the minimum of `32` and `field_len`, in the case of extra data, then computes a mask of 32 entries that are zero at indices `≥ len` and one at indices `< len` by `unsafe_lt_mask`. By multiplying this value to the bytes, it forces all bytes at indices `≥ len` to be zero.
- Packs the 32 bytes into a HiLo instance.
- Selects the header's field with index `header_idx` using indicators and stores the result in `value`.

Special cases are handled separately if the query requests hash, block size, or extra data `len`. They are assigned indices 50, 51, and 52 respectively and `value` selects between the three values with `select_hi_lo`. Booleans `return_hash`, `return_size`, and `return_extra_data_len` are constrained to be true precisely in the respective special case.

The logs-bloom-query case is handled with `handle_logs_bloom`, which will be explained in the discussion of the receipt circuit in section 4.7.

The boolean `return_logs_bloom` is constrained by `handle_logs_bloom` to be true iff $70 \leq$ `field_idx` < 78.

Query is valid if either the index was one of the special-case indices OR the index was in the header field range and lies within the number of fields in the header RLP. This is checked by computing the following values:

- `is_valid_header_idx = (header_idx < header_witness.list_len)`.
  - `is_special_case = return_hash + return_size + return_extra_data_len + return_logs_bloom`.
  - As the values `50`, `51`, and `52` as well as the range `70`, ... ,`77` are mutually exclusive, it is assured that `is_special_case` is either `0` or `1`.
  - `is_valid = is_idx_in_header ? is_valid_header_idx : is_special_case`.
  - Constrains that `is_valid == 1`.

Overall, this function returns the following:

```
PayloadHeaderSubquery {
    header_witness,
    output: AssignedHeaderSubqueryResult {
        subquery: AssignedHeaderSubquery { block_number, field_idx },
        value,
    },
}
```

`virtual_assign_phase0`

- Calls `handle_single_header_subquery_phase0` for each subquery (of its input shard).
- Computes the output commit for the virtual table of key-value pairs `((block_number, field_idx), value)`, where these components are those returned from `handle_single_header_subquery_phase0`.

`handle_single_header_subquery_phase1`

- Calls `EthBlockHeaderChip::decompose_block_header_phase1` on the payload's `header_witness`, thus verifying the remaining second-phase constraints.

`virtual_assign_phase1`

- Calls `handle_single_header_subquery_phase1` for each payload.

We now explain the details of the MMR-related circuit logic.

`assign_mmr`

- Loads all MMR peaks as 32 bytes with range check.
- Computes whether or not the peaks are all zero bytes and collects them as `mmr_bits`.
- Computes `mmr_num_blocks` with `mmr_bits` as a little-endian–bit representation.

`keccak`

- Computes the number of leading zeros in `mmr_bits`.
- Computes the number of actual peaks by `num_peaks = max_num_peaks - num_leading_zeros`.
- Hashes it using `keccak_chip`'s `keccak_var_len` on the concatenated MMR-peak bytes since the number of bytes to actually hash is `32 * num_peaks`.

`mmr_verify`

- Constrains that `list_id < mmr_num_blocks`.
- Computes the number of leading agreeing bits between `mmr_num_blocks` and `list_id`.
- Computes the `peak_id` as `mmr.len() - 1 - num_leading_agree`.
- Computes the intermediate hashes from the Merkle proof verification.
- Takes the `peak_id`-th intermediate hash and MMR peak using indicators.
- Checks that, if proof verification is being done, the intermediate hash and the MMR peak is equal.

4.6 Analysis of the transaction-subquery circuit

We summarize the constraints in the transaction circuit.

The circuit has its output commitment as well as a promise commitment for calls to the block-header subquery as the public instances. The output commitment commits to a virtual table of key-value pairs `((block_number, tx_idx, field_or_calldata_idx), value)`. Here, `block_number` and `tx_idx` are field elements representing a block number and the transaction index, and `field_or_calldata_idx` is a field element that details the query about the transaction. Furthermore, `value` consists of two field elements that jointly encode 32 bytes of data as a HiLo.

`handle_single_tx_subquery_phase0`

Transaction-proof handling

- Loads the transaction proof and the transaction root, then runs the transaction proof with `parse_transaction_proof_phase0`.
- Constrains transaction type to be less than three, which implicitly filters out the case where the MPT proof was an exclusion proof.

Data extraction with `extract_field`

- Assigns and constrains `data_list_index` based on the transaction type using indicators.
- Calls `extract_field` to extract the data field.

Index handling on `field_or_calldata_idx`

- Witnesses `field_or_calldata_idx` and range checks it to be less than 2^{32}.
- Computes `is_idx_in_list = (field_or_calldata_idx < 51)`.
- Computes `field_idx = is_idx_in_list ? field_or_calldata_idx : 1`, so that if the query index is within range, `field_idx` is the original `field_or_calldata_idx` and `1` if otherwise.
- Calls `v2_map_field_idx_by_tx_type` to convert field index to RLP list index (this depends on the transaction type).

Field extraction with `extract_truncated_field`

- Computes an indicator based on the list index.
- Selects the `field_len` and the truncated `SUBQUERY_OUTPUT_BYTES` bytes of the corresponding field element with `select_by_indicator`.
- Sets `len` to be the minimum of actual `field_len` and `SUBQUERY_OUTPUT_BYTES` and constrains that all bytes with index ≥ `len` are trailing zeros.

- Left pads the value to fixed length, unless `field_idx == TX_DATA_FIELD_IDX`, then packs it into a HiLo instance.

Special case handling: Easier cases (`tx_type`, `block_num`, `tx_index`, `data_length`)

- Computes whether the query is for the `tx_type`, `block_num`, `tx_index`, or `data_length` by checking whether `field_or_calldata_idx` equals `0x33`, `0x34`, `0x35`, or `0x38`, respectively.
- Sets `value` to the appropriate HiLo answer via `select_hi_lo`.

Special case handling: `function_selector`

- This computes whether the query is for `function_selector` by checking whether `field_or_calldata_idx` equals `0x36`.
- The goal is to return
  - `TX_CONTRACT_DEPLOY_SELECTOR_VALUE` in the case where it is a contract deployment, so `data_len ≠ 0` and `to_len == 0`;
  - `TX_NO_CALLDATA_SELECTOR_VALUE` when `data_len == 0`;
  - the four-byte selector into a single field element when `data_len ≥ 4` and `to_len ≠ 0`; and
  - to constrain so that the `data_len < 4` and `to_len ≠ 0` case never happens when `function_selector` is the query.
- To do so, the following computation and constraints are applied.
  - Computes `empty_data = (data_len == 0)`.
  - Computes `is_contract_deploy = (1 - empty_data) * (to_len == 0)`.
  - Computes `no_sel = empty_data + is_contract_deploy`, which is equivalent to an OR of two booleans as the two cases are disjoint.
  - Computes `ret1 = is_contract_deploy ? TX_CONTRACT_DEPLOY_SELECTOR_VALUE : TX_NO_CALLDATA_SELECT_VALUE`.
  - Computes `ret2 = bytes_be_to_uint(data[..4])`.
  - Computes `is_valid = (no_sel || (data_len ≥ 4))`.

- Constrains that if the function selector is the query, `is_valid` is true. This forces that in the case where `data_len ≠ 0` and `to_len ≠ 0`, `data_len` must be at least four to answer the function-selector query.
- Returns `no_sel ? ret1 : ret2`, which in conclusion returns `TX_CONTRACT_DEPLOY_SELECTOR_VALUE` when `is_contract_deploy`, `TX_NO_CALLDATA_SELECT_VALUE` when `data_len == 0`, and the four-byte selector when `data_len ≠ 0` and `to_len ≠ 0` (with `data_len ≥ 4` due to additional constraint on `is_valid`).

Special case handling: `data` via `handle_data`

- This computes `in_calldata_range = field_or_calldata_idx in [TX_CALLDATA_IDX_OFFSET = 100, ... , TX_CONTRACT_DATA_IDX_OFFSET = 100000]`.
- This computes `in_contract_data_range = (field_or_calldata_idx ≥ TX_CONTRACT_DATA_IDX_OFFSET)`.
- To compute the correct shift, both the calldata and the contract-data cases are considered separately and the shift selected accordingly.
  - `calldata_shift = field_or_calldata_idx - TX_CALLDATA_IDX_OFFSET`
  - `contract_data_shift = field_or_calldata_idx - TX_CONTRACT_DATA_IDX_OFFSET`
  - `shift = in_calldata_range ? calldata_shift : contract_data_shift`
- In the case of calldata query, the first four bytes need to be ignored. This is handled by selecting the `i`-th byte of the buffer to be either `data_bytes[i + 4]` or `data_bytes[i]` depending on `in_calldata_range` being true or not. In the case where `i + 4 > data_bytes.length()`, the buffer is filled with `(1 - in_calldata_range) * data_bytes[i]` so that zero bytes are added when it is a calldata query.
- The true buffer length is computed as `buffer_len = data_len - 4 * in_calldata_range`.
- This computes `is_valid_calldata = (data_len ≥ 4)`.
- In the case where a calldata query is asked but `is_valid_calldata` is false, the `buffer_len` is set to zero. There was a issue here initially; refer to Finding 3.2. In the fixed version, this is done by computing `buffer_len_is_negative = (1 - is_valid_calldata) * in_calldata_range` so it is 1 when it is a calldata query while

data_len < 4. Then, it overwrites buffer_len = (1 - buffer_len_is_negative) * buffer_len so buffer_len = 0 when buffer_len_is_negative is true.

- This computes the validity of the query by is_in_range = in_calldata_range * is_valid_calldata + in_contract_data_range.
- This sets shift = shift * is_in_range, so on incorrect query, shift = 0.
- This extracts the 32-byte chunk from the buffer by extract_array_chunk_and_constrain_trailing_zeros — this returns is_lt_len, which is a boolean representing if the shift is within the bounds.
- This sets is_in_range = is_in_range * is_lt_len so that if shift is out of bounds, is_in_range is false.
- This packs the 32-byte chunk as a HiLo and returns it alongside is_in_range, which becomes return_data.
- If return_data is true, it is a data query, so value is overwritten by the returned HiLo by utilizing select_hi_lo.

Special case handling: calldata_hash

- Computes whether the query is for calldata_hash by checking field_or_calldata_idx equals 0x37.
- Computes tmp_data_len = data_len * return_calldata_hash so that it is data_len if it is a calldata_hash query, but 0 otherwise.
- Keccak hashes the entire data with keccak_var_len, with tmp_data_len as the buffer length. This avoids hashing in the case the calldata_hash was not queried.

The witness value is overwritten by the computed hash if return_calldata_hash is true by utilizing select_hi_lo.

Final validity check of the query

- Sets is_special_case to the sum of the indicators that the query is for: tx_type, block_num, tx_index, function_selector, calldata_hash, data_length, or data. These can be summed up as they are disjoint cases.
- Constrains that (is_idx_in_list || is_special_case) == 1.

Overall, the function handle_single_tx_subquery_phase0 returns

```
PayloadTxSubquery {
    tx_witness,
    tx_root,
    output: AssignedTxSubqueryResult {
        subquery: AssignedTxSubquery { block_number, tx_idx,
    field_or_calldata_idx },
        value,
    },
}
```

with the components constrained so that under the assumptions that

- the second phase constraints for `tx_witness` hold, and
- the transaction root used in the circuits is the correct transaction root corresponding to the `block_number`,

it holds that the witness `value` is the correct query result represented as a HiLo instance.

`virtual_assign_phase0`

- Calls `handle_single_tx_subquery_phase0` for each subquery (of its input shard).
- Computes the output commit for the virtual table of key-value pairs `((block_number, tx_idx, field_or_calldata_idx), value)`, where these components are those returned from `handle_single_tx_subquery_phase0`.
- Makes a promise call, for each subquery, to the block-header–subquery component to obtain the transaction root for the block number `block_number`. It is constrained that it is equal to the one used for `handle_single_tx_subquery_phase0`.

`handle_single_tx_subquery_phase1`

- Calls `EthTransactionChip::parse_transaction_proof_phase1` on the payload's `tx_witness`, thus verifying the remaining second-phase constraints.

`virtual_assign_phase1`

- Calls `handle_single_tx_subquery_phase1` for each payload.

4.7 Analysis of the receipt-subquery circuit

We summarize the constraints in the receipt-subquery circuit.

The circuit has its output commitment as well as a promise commitment for calls to the header subquery as the public instances. The output commitment commits to a virtual table of key-value pairs `((block_number, tx_idx, field_or_log_idx, topic_or_data_or_address_idx, event_schema), value)`. Here, `block_number` and `tx_idx` are field elements representing a block number and transaction index, and `field_or_log_idx, topic_or_data_or_address_idx` are the field elements representing the detailed query about the log itself. The `event_schema` is a HiLo instance representing the event schema in the EVM. Furthermore, `value` consists of two field elements that jointly encode 32 bytes of data as a HiLo.

The circuit aims to provide the answer for the logs for the `tx_idx`-th transaction at the block number `block_number`. The detailed query is given with two field elements `field_or_log_idx` and `topic_or_data_or_address_idx`. The `event_schema` is checked to be equal to the zeroth (in `0`-index) topic bytes in the case where the query is indeed about the logs. A promise call to the block-header subquery is used to fetch the receipt root — and checks that this receipt root is used for the MPT-proof verification that a receipt exists in the MPT.

Looking into src/components/subqueries/receipt/circuit.rs, we can summarize the constraints introduced by each function as follows.

`handle_single_receipt_subquery_phase0`

We explain this function in multiple parts.

MPT proof verification and index handling

- Loads the receipt proof and MPT root and verifies it with the `parse_receipt_proof_phase0` call.
- Constrains that the corresponding slot is not empty, so the receipt does exist.
- Constrains that `field_or_log_idx` $< 2^{32}$.
- Defines `is_idx_in_list` as `field_or_log_idx < 4 = RECEIPT_NUM_FIELDS`, then computes `field_idx = field_or_log_idx * is_idx_in_list`, so that if the query is actually a field query, then `field_idx = field_or_log_idx`, but if otherwise, `field_idx = 0`.
- Similarly, defines `is_log_idx` as `field_or_log_idx ≥ 100 = RECEIPT_LOG_IDX_O`

FFSET.

- Defines log_idx = (field_or_log_idx - 100) * is_log_idx, so that if the query is actually a log query, then log_idx = field_or_log_idx - 100, but if otherwise, log_idx = 0.
- Checks that log_idx < num_logs where num_logs is the log's length. If this does not hold, set is_log_idx = false and log_idx = 0. This is done by setting is_valid_log_idx = (log_idx < num_logs), then multiplying is_valid_log_idx to both is_log_idx and log_idx.

Field extraction via extract_truncated_field

Now the circuit moves on to fetching relevant data.

- The field_idx corresponds 0 to status, 1 to post state, 2 to cumulative gas, and 3 to log bloom.
- In the actual receipt list, index 0 is post state or status, index 1 is the cumulative gas, and index 2 is the log bloom.
- To handle this difference, the circuit computes
  - get_status as (field_idx == 0),
  - offset = 1 - get_status, and
  - list_idx = field_idx - offset, with an indicator for list_idx.
- Using select_by_indicator, the first 32 bytes of the list_idx-th value are fetched as field_bytes.
- Using select_by_indicator, the field_len of the list_idx-th value are fetched as len.
- It sets len to the minimum of 32 and len.
- The field_bytes are constrained to be all zeros beyond index len.
- In the case of list_idx == 0, the post-state and the status case is determined based on the len. If len < 32, then it should be the status, and if otherwise, it should be the post state. This is done by computing
  - is_post_state_or_status = (list_idx == 0),
  - is_small = (len < 32),

- diff = is_small - get_status, and
- constraining is_post_state_or_status * diff == 0.

If list_idx == 0, this forces (field_idx == 0) == (len < 32) as desired.

- Returns the field_bytes converted to VarLenBytesVec<F> as variable rc_field_bytes.

Log-bloom extraction via handle_logs_bloom

This function is also in the block-header circuit.

- Checks whether the index is a query for the log bloom by checking if field_idx is within [offset, offset+8); denote this as is_offset.
- Sets shift = (field_idx - offset) * is_offset, so that shift is zerp when the query is not log bloom and shift is the correct shift if the query is for the log bloom.
- Takes [shift * 32, shift * 32 + 32) of the logs_bloom_bytes via extract_array_chunk, packs it into HiLo, and returns it alongside is_offset; denote this as logs_bloom_value and is_logs_bloom_idx.

Handling topic_or_data_or_address_idx

- Loads the topic_or_data_or_address_idx as tda_idx.
- Constrains tda_idx $< 2^{32}$.
- Checks whether this index corresponds to topic or data.

For topic,

- Sets is_topic = (tda_idx < 4) * is_log_idx so is_topic is true iff tda_idx < 4 and it is a log query.
- Sets topic_idx = tda_idx * is_topic so it is tda_idx if is_topic is true and 0 otherwise.

For data,

- Sets is_data_idx = (tda_idx ≥ 100) * is_log_idx so is_data_idx is true iff tda_idx ≥ 100 and it is a log query.

- Sets `data_idx = (tda_idx - 100) * is_data_idx` so it is `tda_idx - 100` if `is_data_idx` is true and `0` otherwise.

Both `is_topic` and `is_data_idx` are modified later after fetching the topic and data values — they are set to zero if the corresponding values are invalid.

Log parsing with `extract_receipt_log` and `conditional_parse_log`

In the version that fixes Finding 3.1, `conditional_parse_log` has been renamed `conditional_parse_log_phase0`.

- The witness `extract_receipt_log` is defined in `EthReceiptChip`.
- This uses an indicator corresponding to `log_idx` and selects the bytes and length for the `log_idx`-th log using `select_by_indicator`.
- This log is parsed through `conditional_parse_log` along with the parsing flag `is_log_idx`, so the parsing is only done when `is_log_idx` is true.
- The log is replaced with a dummy log if `is_log_idx` is false — then the log is parsed with the `parse_log` function.
- The `parse_log` function decomposes the RLP array with the `RlpChip`'s `decompose_rlp_array_phase0` to get `address, topics, data`.
- The `topics` is once again RLP decomposed to get the `topics_list`.

Fetching topics, data, and address

- The data is fetched through `extract_data_section`:
  - It calls `extract_array_chunk_and_constrain_trailing_zeros`.
  - It returns `32 * data_idx < data_len` as `is_valid`.
  - This `is_valid` is multiplied at `is_data_idx`.
- The topic is fetched by `select_array_by_indicator` on `topic_bytes` with an indicator based on `topic_idx`:
  - It computes `is_valid_topic = (topic_idx < num_topics)`.
  - It sets `is_topic = is_topic * is_valid_topic`.
- The address is fetched from `address`.

Event-schema constraints

- Sets `no_constrain_event = (event_schema == zero bytes)`.
- Sets `event_diff = topic_bytes[0] - event_schema`.
- Sets `event_eq = (event_diff == zero_bytes)`.
- Constrains that `no_constrain_event || (event_eq && is_log_idx)` is true.

So when `event_schema` is nonzero, `event_schema` must be `topic_bytes[0]` and `is_log_idx` must be true.

Special case handling with indicators

- Checks if the query is `tx_type`, `block_num`, or `tx_idx` by checking if `field_or_log_idx` is `0x32`, `0x33`, or `0x34`.
- Checks if the query is `address` by checking if `tda_idx = 0x32` and `is_log_idx` is true.
- Checks if the query is sound by summing `is_idx_in_list, is_tx_type, is_block_num, is_tx_idx, is_logs_bloom_idx, is_topic, is_addr, is_data_idx` and constraining it to be equal to `1`.
- Gathers all the query answers for each cases, turns them into HiLo, and selects them based on the indicator. Here, the field element is additionally handled with `prep_field`.
- Turns the `event_schema` into bytes, range checks them, and then packs it into HiLo.

Field element handling with `prep_field`

- This function exists to pad receipt field elements to 32 bytes appropriately.
- The constant array `left_pad_indicator` contains whether a field element should be padded left or not.
- Selects whether to left pad by selecting from `left_pad_indicator` with `select_from_idx` and denotes it as `left_pad`.
- Selects either the left-padded byte array or the original byte array based on

left_pad.

- Packs the resulting byte array into HiLo.

Summary

Overall, the function handle_single_receipt_subquery_phase0 returns[2]

```
PayloadReceiptSubquery {
        rc_witness,
        rc_root,
        output: AssignedReceiptSubqueryResult {
            subquery: AssignedReceiptSubquery {
                block_number,
                tx_idx,
                field_or_log_idx,
                topic_or_data_or_address_idx: tda_idx,
                event_schema,
            },
            value,
        },
    }
```

with the components constrained so that under the assumptions that

- all the second-phase constraints hold, and
- the receipt root used for the MPT proof verification is the correct receipt root corresponding to the block number,

it holds that the value is the HiLo instance that corresponds to the desired query result.

virtual_assign_phase0

- Calls handle_single_receipt_subquery_phase0 for each subquery (of its input shard).
- Computes the output commit for the virtual table of key-value pairs ((block_number, tx_idx, field_or_log_idx, topic_or_data_or_address_idx,

[2] In the version fixing Finding 3.1, log_witness, is returned as well.

event_schema), value), where these components are those returned from handle_single_receipt_subquery_phase0.

- Makes, for each subquery, a promise call to the block-header-subquery component to obtain the receipt root at the block block_number.

handle_single_receipt_subquery_phase1

- Calls EthReceiptChip::parse_receipt_proof_phase1 on the payload's rc_witness, thus verifying the remaining second-phase constraints with the exception of those discussed in Finding 3.1[3].

virtual_assign_phase1

- Calls handle_single_receipt_subquery_phase1 for each payload.

[3] In the fixed version, conditional_parse_log_phase1 is called as well, handling the previously missing second-phase constraints.

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 two findings. No critical issues were found. One finding was of high impact and one was of low impact. 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.