Zellic

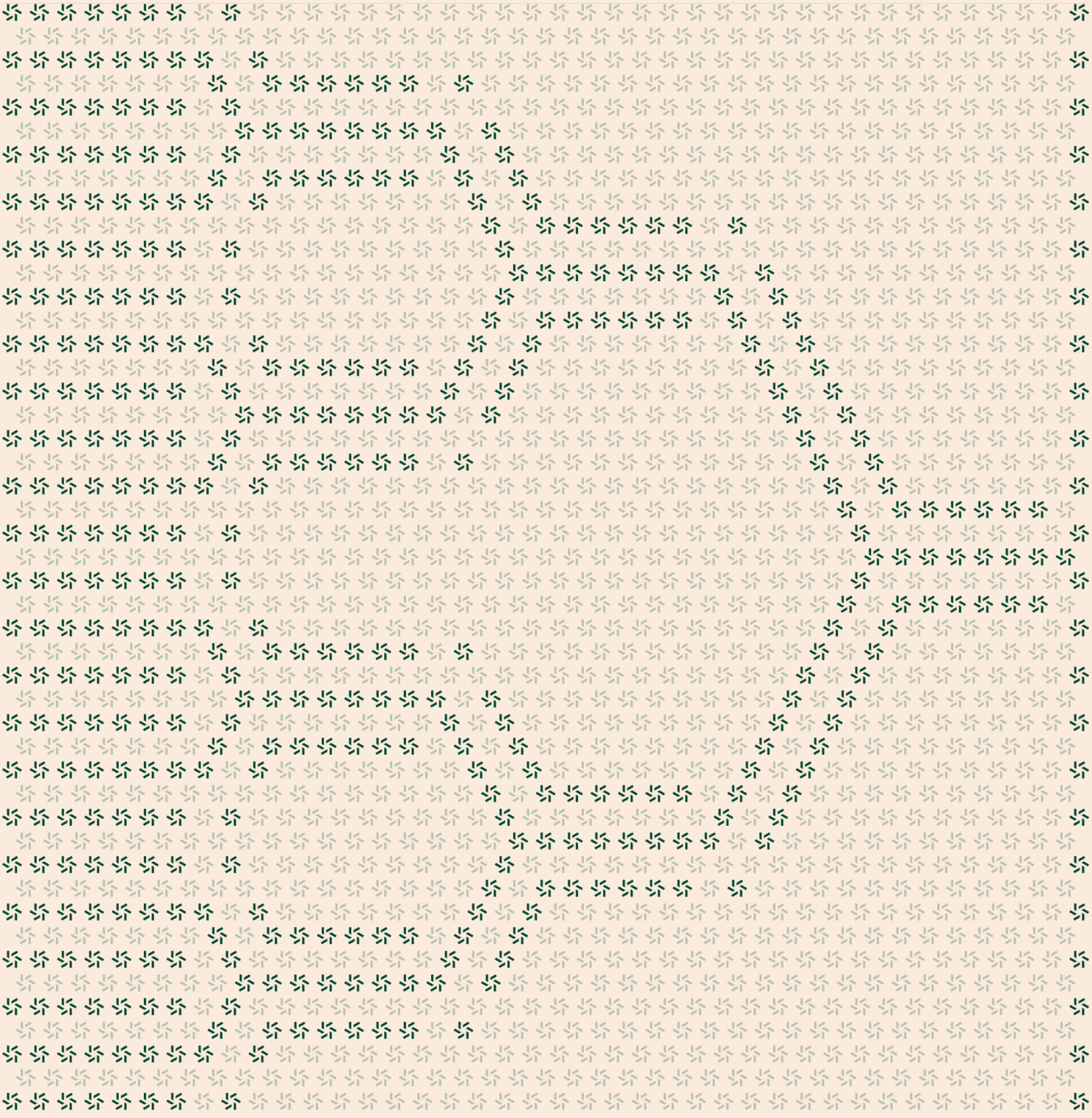
Prepared for
Duncan Tebbs
Nebra

Prepared by
Malte Leip
Sylvain Pelissier
Zellic

August 28, 2024

UPA circuits circuitId hash function change

Zero Knowledge Security Assessment



Contents

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for Nebra from August 8th to August 12th, 2024. During this engagement, Zellic reviewed the scope's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. 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 and the client. In this assessment, we sought to answer the following questions:

- Do the code changes correctly implement the changes made to the specification regarding computation of the `circuitId` using Keccak instead of Poseidon?
- Do the changes introduce any soundness or completeness issues?

1.3. 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
- Tests
- Code related to the fixed circuits

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

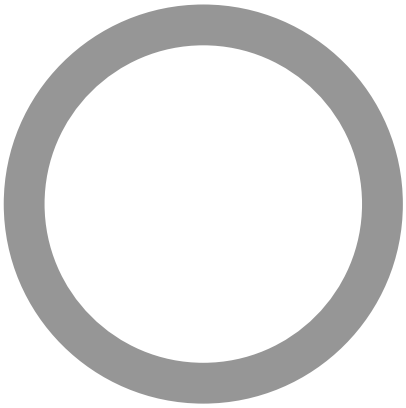
1.4. Results

During our assessment of the scoped circuits, we discovered three findings, all of which were informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for Nebra's benefit in the Discussion section (4. ↗).

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	0
Low	0
Informational	3



2. Introduction

2.1. About UPA circuits circuitId hash function change

Nebra contributed the following description of UPA circuits circuitId hash function change:

> NEBRA's Universal Proof Aggregator (UPA) v1.0.0 reduces the on-chain verification costs of Groth16 proofs by aggregating them into a single proof.

2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses 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 circuit. While rare, it is possible that a circuit is overconstrained. In this case, appropriately assigning witnesses 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 is 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, guidelines, and code quality standards.

For each finding, Zellic assigns 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.

Zellic organizes 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.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped circuits itself. These observations — found in the Discussion (4. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

UPA circuits circuitId hash function change

Type	Rust
Platform	halo2
Target	saturn
Repository	https://github.com/NebraZKP/saturn ↗
Version	`Diff from 8d5e5ad0cbe514db607b4ce30e090c482cca2d33 to a76a846595e0f0b483c790f0f5738c34a9199fb8`

Programs	`batch_verify/common/chip.rs`
	`batch_verify/universal/chip.rs`
	`batch_verify/universal/mod.rs`
	`batch_verify/universal/types.rs`
	`keccak/chip.rs`
	`keccak/inputs.rs (KeccakVarLenInput only, except sample functions)`
	`keccak/mod.rs`
	`keccak/multivar.rs`
	`keccak/utils.rs (g1_point_limbs_to_bytes, g2_point_limbs_to_bytes, multi_coordinates_to_bytes)`
	`keccak/variable.rs`
	`outer/universal.rs`
	`utils/commitment_point.rs`
	`utils/hashing.rs`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of three person-days. The assessment was conducted by two consultants over the course of one calendar week.

Contact Information

The following project manager was associated with the engagement:

Jacob Goreski
Jr. Engagement Manager
jacob@zellic.io

Chad McDonald
Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Malte Leip
Engineer
malte@zellic.io

Sylvain Pelissier
Engineer
sylvain@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

August 8, 2024	Start of primary review period
August 12, 2024	End of primary review period

3. Detailed Findings

3.1. Range check in `add_slice_at_offset`

Target	KeccakMultiVarHasher		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Informational

Description

In circuits/src/keccak/multivar.rs, the function `KeccakMultiVarHasher::absorb_var` checks that the length of the input fits in `MAX_BIT_SIZE` bits but nothing stricter:

```
assert!(input.len() < 1 << MAX_BIT_SIZE, "input too long");
```

So a length of `(1 << MAX_BIT_SIZE) - 1` is in principle possible.

Then, in `add_slice_at_offset`, there is the following snippet:

```
// Require `len <= padded input length`
let padded_input_len = source.len();
let padded_input_len_plus_one =
    ctx.load_constant(F::from(padded_input_len as u64 + 1));
range.range_check(ctx, len, MAX_BIT_SIZE);
range.check_less_than(ctx, len, padded_input_len_plus_one, MAX_BIT_SIZE);
```

Here, `source` is one of the `input` vectors from `absorb_var`, so `padded_input_len` can in principle have value `(1 << MAX_BIT_SIZE) - 1`, and hence `padded_input_len_plus_one` can have value `1 << MAX_BIT_SIZE`. This is a value that does not fit in `MAX_BIT_SIZE` bits, which is an assumption required for the soundness of `check_less_than`, however.

Impact

The range check

```
range.check_less_than(ctx, len, padded_input_len_plus_one, MAX_BIT_SIZE);
```

is unsound in the case of inputs of the very specific padded length for a variable length input of `(1 << MAX_BIT_SIZE) - 1`. Note that a typical attacker cannot control this padded length, which is determined at verification-key–generation time. It seems very unlikely that the circuit will be instantiated with a variable length input of this length on accident, in particular due to `MAX_BIT_SIZE` being 32, so

that this would amount to a quite large variable-length input.

Assuming a variable-length input with padded length `(1 << MAX_BIT_SIZE) - 1`, we cannot assume that the above range check constrains `len` to be smaller than `padded_input_len_plus_one = 1 << MAX_BIT_SIZE`. This is as the assumption that `padded_input_len_plus_one` fits in `MAX_BIT_SIZE` bits fails. However, as `len` is potentially able to be chosen by the attacker, that it fits in `MAX_BIT_SIZE` bits *is* checked with an explicit range check in the line right before:

```
range.range_check(ctx, len, MAX_BIT_SIZE);
```

This ensures that `len < (1 << MAX_BIT_SIZE)`. Thus, the intended inequality `len < padded_input_len_plus_one` still holds.

Recommendations

Due to the reasoning outlined in the impact section, the code is not vulnerable as is, due to the separate range check on `len`. However, for best practice, we recommend to use a slightly stricter check on the padded input length in the function `KeccakMultiVarHasher::absorb_var`:

```
assert!(input.len() < 1 << MAX_BIT_SIZE, "input too long");
assert!(input.len() < (1 << MAX_BIT_SIZE) - 1, "input too long");
```

This will ensure that `padded_input_len_plus_one` is at most `MAX_BIT_SIZE` bits wide.

Remediation

This issue has been acknowledged by Nebra, and a fix was implemented in commit `3d3567ad` ↗.

3.2. Incorrect deprecated `proofId` computation in the fixed-length case

Target	KeccakCircuit		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

In circuits/src/keccak/mod.rs, the deprecated function `KeccakCircuit::compute_proof_id_fixed_length` is intended to compute the `proofId`. However, the calculation is incorrect, as it is missing the `circuitId`.

Impact

The result of `KeccakCircuit::compute_proof_id_fixed_length` is incorrect. This function is intended for the fixed-length circuits, which are not intended to be used anymore.

Recommendations

Marking a function as `#[deprecated]` only causes the compiler to issue a warning on use. If the function is incorrect and will not be fixed, we recommend to cause it to panic (as is done in other places in the fixed-length case) or to remove it.

Remediation

This issue has been acknowledged by Nebra. Nebra removed the function from the circuit in the public repository.

3.3. Mixing limbs from different variable-length inputs in KeccakMultiVarHasher

Target	KeccakMultiVarHasher		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The KeccakMultiVarHasher (in circuits/src/keccak/multivar.rs) is intended to compute the Keccak hash of a concatenation of first zero or more fixed-length inputs, given in bytes, and zero or more variable-length inputs, which are assumed to be field elements, represented by `NUM_LIMBS` limbs. Usage involves the caller passing a circuit variable `len` for each of the variable-length inputs, which determines how much of the variable-length input (which comes in padded form) to hash. The value `len` is the number of *limbs* to use, not the number of field elements those limbs represent. It is checked in circuit that the sum of `len` for the various variable-length inputs is divisible by `NUM_LIMBS`. However, there is no check that each of the individual values of `len` are divisible by `NUM_LIMBS`. Thus, it is possible to mix limbs from multiple variable-length inputs. For example, only the first limb from `NUM_LIMBS` different variable-length inputs could be used, with those `NUM_LIMBS` limbs being interpreted as a single field element. This is not intended usage.

The way KeccakMultiVarHasher is used in the remainder of the codebase ensures in each case that each value of `len` is in fact divisible by `NUM_LIMBS`.

Impact

The KeccakMultiVarHasher can be used in an unintended way that is not prevented by KeccakMultiVarHasher itself. There is no impact for the reviewed codebase itself, however, as the unintended usage cannot occur from the actual callers.

Recommendations

We recommend to add a comment to the documentation comments for KeccakMultiVarHasher explaining that the lengths being multiples of `NUM_LIMBS` is not checked and that the caller must ensure this.

Remediation

This issue has been acknowledged by Nebra.

4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1. Inaccuracies in the specification

We collect here some inaccuracies in the specification spec/circuits/var_len_keccak.md that we encountered.

Example for multi-variable length query

In the example for the multi-variable length query, under Step 2, the matrix is missing two rows of zero at the bottom. The end result (the line before the next heading "Slice-Adder Matrix") is missing the 9 and a 0.

Reuse of variable k

In the "Slice-Adder Matrix" section, the slice-adder matrix S_k is discussed, making k bound. However, k is reused when describing the number of columns of S_k as $\sum_k L_k$, reusing k. Using a different variable in the sum would be better notation.

Keccak-circuit domain-tag calculation

The specification for the Keccak circuit (spec/circuits/var_len_keccak.md) does not accurately describe how domain tags are used after the changes made to the code. The specification describes as Step 3 conversion to a field element, which was relevant for use in the Poseidon hash function but is not relevant anymore (and hence not reflected in the code) after the switch to Keccak for `circuitId` calculation. We also noticed and reported this in our audit whose first draft was delivered August 8th 2024, under section 4.9.

Out-of-scope aspects remarked in earlier audits

This version of the spec still uses inaccurate LegoSNARK terminology for the gnark Groth16 extension. For more details, see section 4.2 in the report from June 7th. The specification also reflects a hash-to-field function that we discussed in 3.12 and 3.13 in our report from August 8th 2024.

4.2. Possible optimization in `slice_adder_column`

For computation of multi-variable length queries in circuits/src/keccak/multivar.rs, a significant part of the logic involves computation of the slice-adder matrix S_k. This matrix has L_k rows and $\sum_i L_i$ columns. The values of L_i are constants (known during circuit generation). There are also circuit variables $0 \leq \ell_i \leq L_i$. The matrix S_k is now defined as follows:

$$o_k = \sum_{i=0}^{k-1} \ell_i$$

$$(S_k)_{i,j} = \begin{cases} 1 & \text{if } j - i = o_k \text{ and } i < \ell_k \\ 0 & \text{else} \end{cases}$$

To make an example, let us consider the case where $L_0 = 2, L_1 = 3, L_2 = 4$ and we are considering S_1. Then, o_1 will satisfy $0 \leq o_1 \leq 2$. Should we have $o_1 = 0$ and $\ell_1 = 3$, then the matrix would look as follows:

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

If we had $\ell_1 < 3$, then only the first ℓ_1 entries with 1 in the above matrix (counting them from the top) would contain 1, the rest would contain 0 instead.

If instead we have $o_1 = 2$ and $\ell_1 = 3$, then the matrix would look as follows:

$$\begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

In general, the following matrix shows which entries *could* be either 0 or 1, depending on what the values of the ℓ_i are, marked by $?$.

$$\begin{pmatrix} ? & ? & ? & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & ? & ? & ? & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & ? & ? & ? & 0 & 0 & 0 & 0 \end{pmatrix}$$

Note that various entries are always zero. These come in three types:

1. Columns to the right
2. A triangle at the bottom left
3. A triangle at the top right (but left of the already mentioned columns)

In general, the entries of these three classes can be described as follows:

1. $j \geq \sum_{i=0}^{k} L_i$
2. $j - i < 0$
3. $j - i > \sum_{i=0}^{k-1} L_i$ but not $j \geq \sum_{i=0}^{k} L_i$

Note that the following inequalities always hold:

$$0 = \sum_{i=0}^{k-1} 0 \leq \sum_{i=0}^{k-1} \ell_i \leq \sum_{i=0}^{k-1} L_i$$

Thus, $0 \leq o_k \leq \sum_{i=0}^{k-1} L_i$. We can now show that the three types of entries above must always be zero:

1. As $0 \leq i < L_k$, the inequality $j \geq \sum_{i=0}^{k} L_i$ implies $j - i > \sum_{i=0}^{k-1} L_i = o_k$.
2. As $o_k \geq 0$, the inequality $j - i < 0$ directly implies $j - i \neq o_k$.
3. If $j - i > \sum_{i=0}^{k-1} L_i$, then by definition $j - i > o_k$.

Instead of using constraints to calculate the matrix entries from their definition, it is more efficient to use the constant 0 where the entry must be zero.

This optimization is done in the code for the first two types of entries that are always zero.

For the first type, this is handled by `prepare_preimage` and `add_slice_at_offset` as follows: `prepare_preimage` passes $\sum_{i=0}^{k} L_i$, as `max_nonzero_position` to `add_slice_at_offset`, and then `add_slice_at_offset` never calculates columns from `max_nonzero_position` forward.

For the second type, this is handled in `slice_adder_column`. This does not require any additional data; the entries with $i > j$ can directly be filled with the constant zero.

The optimization not yet implemented is the third type. This could be done as follows.

The `prepare_preimage` could, in addition to $\sum_{i=0}^{k} L_i$, also pass $\sum_{i=0}^{k-1} L_i$ to `add_slice_at_offset`, which then passes this value further to `slice_adder_column`, perhaps as an additional argument `offset_max`. Then those entries for which `j > i + offset_max` could be set to the constant zero rather than constraining their calculation from the definition.

The amount of constraints saved by this optimization would be similar to the constraints saved by the second optimization.

4.3. Usage of named constants

In the function `prepare_preimage` in circuits/src/keccak/multivar.rs, there are some places in which literal constants are used instead of defined named constants that are intended for the respective use case. For best practice, we would recommend to use the named constants in those cases.

Some examples are present in the following snippet,

```
let mut preimage = Vec::with_capacity(
    self.fixed_input.len() + var_preimage_len / 3 * 32,
);
preimage.extend_from_slice(&self.fixed_input);
let var_bytes = multi_coordinates_to_bytes(ctx, range, &preimage_var);
assert_eq!(var_bytes.len(), var_preimage_len / 3 * 32);
preimage.extend_from_slice(&var_bytes);
```

where it would be more consistent to use `NUM_LIMBS` instead of 3 (as is done elsewhere in the file).

Similarly for

```
let three = ctx.load_constant(F::from(3u64));
```

where `NUM_LIMBS` could also be used.

Finally, in the following snippet, one could use `NUM_BYTES_FQ`, which is also used elsewhere, instead of `32u64`.

```
let thirty_two = ctx.load_constant(F::from(32u64));
```

4.4. Additional assert

In circuits/src/keccak/utils.rs, the function `assigned_field_element_bytes_into_parts` is implemented as follows:

```
/// Splits the field element represented by `bytes` into `NUM_PARTS` parts.
fn assigned_field_element_bytes_into_parts<'a, F, const NUM_PARTS: usize>(
    ctx: &mut Context<F>,
    chip: &RangeChip<F>,
    bytes: impl ExactSizeIterator<Item = &'a AssignedValue<F>>,
) -> [AssignedValue<F>; NUM_PARTS]
where
    F: EccPrimeField,
```

```
{
    let num_bytes = num_bytes::<F>();
    assert_eq!(num_bytes, bytes.len(), "Wrong number of bytes");
    let num_bytes_in_part = num_bytes / NUM_PARTS;
    let powers = byte_decomposition_powers()
        .into_iter()
        .take(num_bytes_in_part)
        .map(|power| QuantumCell::from(ctx.load_constant(power)))
        .collect_vec();
    bytes
        .into_iter()
        .chunks(num_bytes_in_part)
        .into_iter()
        .map(|chunk| {
            chip.gate.inner_product(
                ctx,
                chunk.into_iter().cloned().map(QuantumCell::from),
                powers.clone(),
            )
        })
        .collect_vec()
        .try_into()
        .expect("Conversion from vector into array is not allowed to fail")
}
```

This function is passed a certain number of field elements, and attempts to convert chunks of `num_bytes_in_part` field elements at a time to field elements, by interpreting them as the byte representation of the value in little endian.

It appears that the intention was that the length of the input iterator would be `NUM_PARTS * num_bytes_in_part` or phrased differently that `NUM_PARTS` divides `num_bytes`. Should this not be the case, then the result will, with the current implementation, correspond to the result with the appropriately zero extended input. The zero extension would be by appending zeros, which does not change the value represented in the case of little-endian format.

However, this behavior depends on the implementation of the halo2's flex gate, whose `inner_product` function is documented as follows:

```
/// Constrains and returns the inner product of `<a, b>`.
///
/// Assumes 'a' and 'b' are the same length.
/// * `ctx`: [Context] to add the constraints to
/// * `a`: Iterator of [QuantumCell] values
/// * `b`: Iterator of [QuantumCell] values to take inner product of `a` by
```

Thus, we recommend to consider not relying on the current behavior of `inner_product`. If as-

`signed_field_element_bytes_into_parts` is not needed to be used for input that is not full length, then it could be asserted that `NUM_PARTS` divides `num_bytes`:

```
let num_bytes = num_bytes::<F>();
assert_eq!(num_bytes, bytes.len(), "Wrong number of bytes");
let num_bytes_in_part = num_bytes / NUM_PARTS;
assert_eq!(num_bytes_in_part * NUM_PARTS, num_bytes, "Number of bytes not
    divisible by number of parts");
```

4.5. Inefficient vector capacity

In the function `keccak_inputs_from_ubv_instances` in the file circuits/src/keccak/utils.rs, there is the following vector defined with capacity:

```
let mut keccak_inputs =
    Vec::with_capacity(ubv_instances.len() * inputs_per_proof);
```

However, `keccak_inputs` will ultimately have `ubv_instances.len() * inner_batch_size` elements. Thus, if `inputs_per_proof < inner_batch_size`, additional allocations will need to be performed. If instead `inputs_per_proof > inner_batch_size`, then an unnecessary amount of memory will be allocated for this vector. We thus recommend to change the allocated capacity:

```
let mut keccak_inputs =
    Vec::with_capacity(ubv_instances.len() * inputs_per_proof);
    Vec::with_capacity(ubv_instances.len() * inner_batch_size);
```

4.6. Inaccurate comments

In some reviewed places, we noticed inaccurate comments. We collect them here.

Function `KeccakPaddedCircuitInput::to_instance_values`

In circuits/src/keccak/mod.rs, the function `KeccakPaddedCircuitInput::to_instance_values` has the following documentation comment:

```
/// For fixed-len: Appends the vk hash to the application public inputs.
```

```
/// For variable-len: Appends the length and the vk hash to the application
/// public inputs.
pub fn to_instance_values(&self) -> Vec<F> {
    /*
    let mut result = if self.is_fixed {
        vec![self.app_vk_hash]
    } else {
        vec![self.len, self.app_vk_hash]
    };
    */
    assert!(self.is_var(), "Fixed length keccak no longer supported");
    let mut result = vec![self.len];
    result.extend_from_slice(&self.app_vk.flatten());
    result.push(self.has_commitment);
    result.push(self.commitment_hash);
    result.extend_from_slice(&self.commitment_point_limbs);
    result.extend_from_slice(&self.app_public_inputs);
    result
}
```

This comment is inaccurate in the following ways:

- In sentences of the form "Appends X to Y", X and Y are switched. We also pointed this out in section 4.1 of the audit from May 20th, 2024.
- There is a description for fixed-length case, but that case is actually disabled.
- The variable-length description is not updated to use the verification key rather than its hash, as well as the inclusion of the commitments.

Function `AssignedKeccakInput::from_keccak_padded_input`

Also in circuits/src/keccak/mod.rs, the function `AssignedKeccakInput::from_keccak_padded_input` includes the following snippet:

```
// Constrain `len + has_commitment < MAX_LEN`
let len_inputs_and_commitment =
    range.gate.add(ctx, len, has_commitment);
range.check_less_than_safe(ctx, len_inputs_and_commitment, max_len + 1);
```

The check is for `len + has_commitment <= max_len`, which is the correct check — `len` is the number of public inputs without the hashed commitment, the number of hashed commitments is added to that, and the sum should be at most the total number of padded public inputs. The comment uses < instead of <=, however.

4.7. Improvements possible to make `KeccakCircuit::new` more robust

In circuits/src/keccak/mod.rs, the function `KeccakCircuit::new` contains the following code:

```
for input in inputs.0 {
    let assigned_input = AssignedKeccakInput::from_keccak_padded_input(
        ctx, &range, input,
    );
    let circuit_id = Self::compute_circuit_id(
        ctx,
        &range,
        &mut keccak,
        &assigned_input,
    );
    if is_fixed {
        Self::compute_proof_id_fixed_length(
            ctx,
            &range,
            &mut keccak,
            &assigned_input,
        );
        panic!("Keccak fixed length no longer supported");
    } else {
        // Specification: Proof ID Computation
        Self::compute_proof_id(
            ctx,
            &range,
            &mut keccak,
            &circuit_id,
            &assigned_input,
        );
    }
    // Specification: Curve-to-Field Hash
    Self::commitment_point_hash_query(
        ctx,
        &range,
        &mut keccak,
        &assigned_input,
    );
    public_inputs.push(assigned_input);
}

// Specification: Final Digest Computation.
// Here we select only the even var_len_queries because those
// contain the proofIds. The odd ones contain the circuitIds
// which are not hashes into the final digest.
```

```
let intermediate_outputs = keccak
    .var_len_queries()
    .iter()
    .skip(1)
    .step_by(2) // we skip the circuitId computations
    .flat_map(|query| query.output_bytes_assigned().to_vec())
    .collect::<Vec<_>>();
```

The logic to extract the `proofId`s towards the end of this snippet, involving skipping the first item and then skipping every second item, relies on the following:

- The function `compute_circuit_id` creates exactly one var len query each.
- We are in the variable-length case (as the fixed-length case panics), so `compute_proof_id` is used instead of `compute_proof_id_fixed_length`.
- The function `compute_proof_id` creates exactly one var len query, and the result of that query is the `proofId` we would like to extract.
- The function `commitment_point_hash_query` does not create any var len query (only a fixed len one).

While the outlined logic is correct, it would be more robust to handle the `proofId`s the same way as the `circuitId`s. In the case of `KeccakMultiVarHasher::finalize`, the result of the query is extracted and returned immediately,

```
keccak_chip.keccak_var_len(ctx, range, preimage, len);
return keccak_chip
    .var_len_queries()
    .last()
    .expect("no queries")
    .output_bytes_assigned()
    .to_vec();
```

and the functions `KeccakChip::multi_var_query` and `KeccakCircuit::compute_circuit_id` forward this return value.

One could do something analogous in `compute_proof_id` (that is, extract the result immediately after creating the query and returning it), and then in `KeccakCircuit::new`, one could push the return value to `intermediate_outputs` instead of needing to rely on the precise query order to obtain it at the end. This would make the code less likely to break and require changes should some of the called functions be changed to create more or less var len queries. It would also make it easier to see that the code is correct, not requiring to check the above four bullet points.

5. Assessment Results

At the time of our assessment, the reviewed code was not deployed to the Ethereum Mainnet.

During our assessment on the scoped circuits, we discovered three findings, all of which were informational in nature.

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, of course, also cannot make guarantees about any code added to the project after the version reviewed during 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, Zellic provides 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. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

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.