Zellic

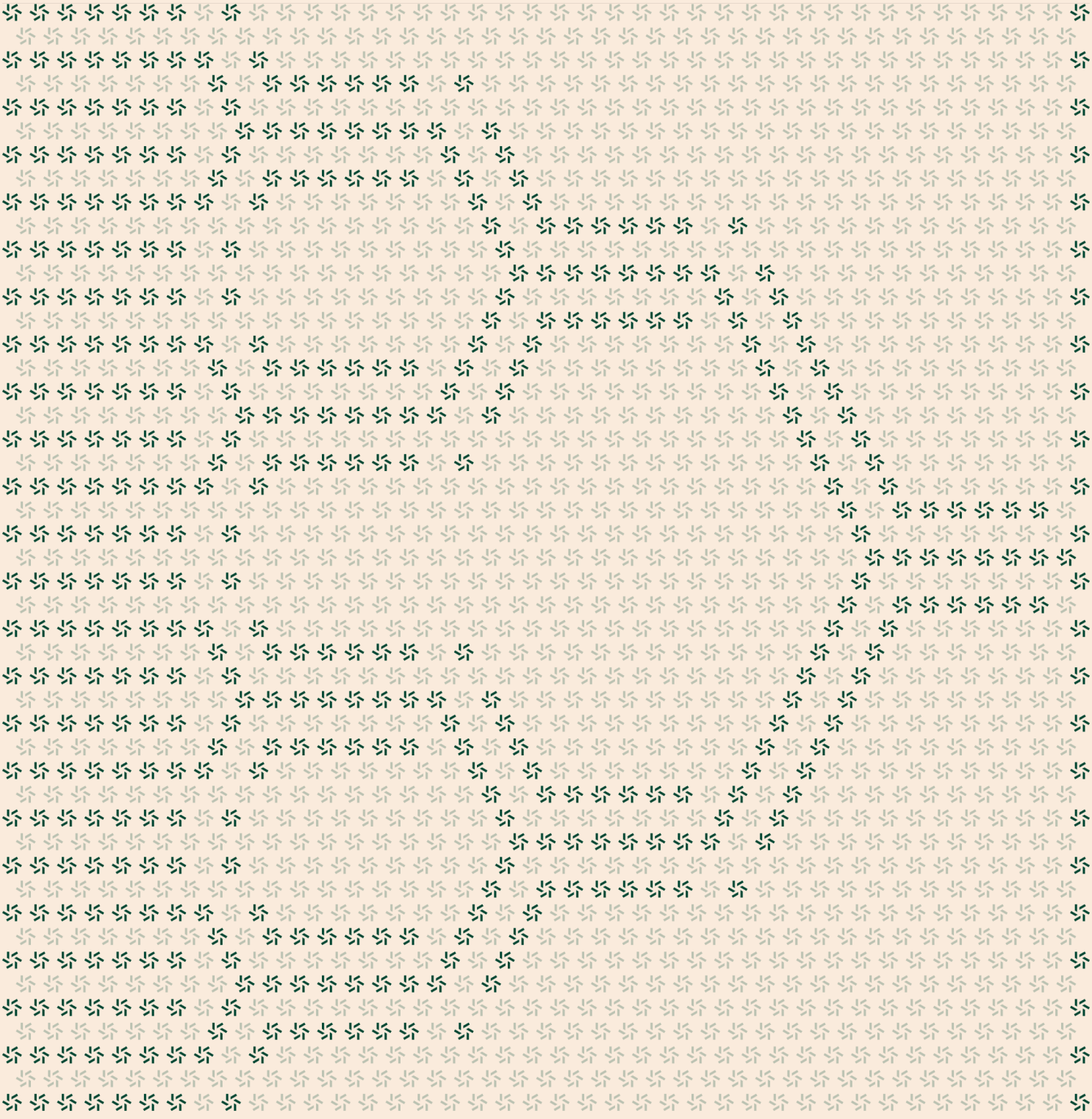
Prepared for
Shumo Chu
Nebra

Prepared by
Malte Leip
Sylvain Pelissier
Mohit Sharma
Zellic

May 20, 2024

Universal Proof Aggregator

Proof Circuit 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 April 15th to May 3rd, 2024. During this engagement, Zellic reviewed Universal Proof Aggregator'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:

- Are there any underconstrained variables in the circuits?
- Are there bugs related to the improper use of dependencies, particularly the halo2 libraries and their forks?
- Is there anything that may threaten the completeness or soundness of the overall protocol?

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
- Key custody

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

The out of scope base Keccak base circuit is a modified forked version of the Keccak circuit from Axiom's halo2-lib repository. For neither version of the base Keccak circuit was detailed specification or documentation available, which prevented us from fully checking correct usage.

Similarly, lack of documentation prevented us from fully verifying correctness of the `gen_outer_evm_verifier` function in `circuits/src/outer/utils.rs`.

1.4. Results

During our assessment on the scoped Universal Proof Aggregator circuits, we discovered six findings. No critical issues were found. One finding was of high impact, one was of medium impact, and the remaining findings were informational in nature.

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

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	1
Medium	1
Low	0
Informational	4

2. Introduction

2.1. About Universal Proof Aggregator

Nebra contributed the following description of Universal Proof Aggregator:

> 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 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 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:

Universal Proof Aggregator Circuits

Repository	https://github.com/NebraZKP/Saturn ↗
Version	Saturn: `83de216e8abbd186026a0ed7ad56e1427245d0a8`
Programs	• circuits/src/batch_verify/*.rs • circuits/src/keccak/*.rs • circuits/src/outer/*.rs • circuits/src/utils/bitmask.rs • circuits/src/utils/hashing.rs • Detailed scope in circuits/README.md
Type	Rust
Platform	Halo2

2.4. Project Overview

Zellic was contracted to perform a security assessment with three consultants for a total of three and a half 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

Sylvain Pelissier
Engineer
sylvain@zellic.io

Mohit Sharma
Engineer
mohit@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

April 15, 2024	Kick-off call
April 15, 2024	Start of primary review period
May 3, 2024	End of primary review period

3. Detailed Findings

3.1. Verification-batching implementation unsound

Target	UniversalBatchVerifierChip		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	Medium	**Impact**	High

Summary

The universal batch verifier verifies a batch of Groth16 ↗ proofs. Instead of verifying each proof individually, they are combined so that only an equality needs to be checked, which is more efficient. This finding concerns the assumptions under which this combined check implies validity of all the individual proofs; the current codebase violates these assumptions in a subtle manner, making the usual security proof for the batched check inapplicable. It is plausible that this could also in practice allow an attacker to construct a batch including an invalid proof for a legitimate (not chosen by the attacker) verification key but with UniversalBatchVerifierChip still verifying the batch.

Review of how Groth16 proofs are verified

To describe this finding, let us begin by reviewing how individual Groth16 proofs are verified and how the batching process works abstractly, following the specification in `spec/circuits/universal_batch_verifier.md`.

We are going to mostly follow the notational conventions from the specification and denote by $e\colon \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ the pairing over the BN254 elliptic curve. In contrast to the specification, we will write $\mathbb{G}$ and $\mathbb{G}_2$ but also $\mathbb{G}_T$ additively. The scalar field of BN254 will be denoted by $\mathbb{F}_r$.

A Groth16 circuit comes with a *verification key*, which consists of $4 + (L + 1)$ elements of the groups $\mathbb{G}_1$ and $\mathbb{G}_2$, where L is the number of public inputs.[1] Concretely, a verification key is of the form $vk = (\alpha, \beta, \gamma, \delta, (s_0, \ldots, s_L)) \in \mathbb{G}_1 \times \mathbb{G}_2 \times \mathbb{G}_2 \times \mathbb{G}_2 \times \mathbb{G}_1^{\times L+1}$.

An instance must then provide the L public inputs, which are elements $P_1, \ldots, P_L \in \mathbb{F}_r$.

A proof that proves knowledge of witnesses to the circuit with given public inputs then consists of only three group elements: $\pi = (A, B, C) \in \mathbb{G}_1 \times \mathbb{G}_2 \times \mathbb{G}_1$.

A proof is valid if it satisfies the following pairing check,

$$e\left(A, B\right) = e\left(\alpha, \beta\right) + e\left(\sum_{i=0}^{L} P_i \cdot s_i, \gamma\right) + e\left(C, \delta\right)$$

[1] As it is not relevant for this finding, we are going to ignore that some proofs to be verified might be for circuits with less public inputs, which are thus padded.

where $P_0 = 1$. By defining $S = \sum_{i=0}^{L} P_i \cdot s_i$ and rearranging the above equation, we can equivalently check the following.

$$e\left(-A, B\right) + e\left(\alpha, \beta\right) + e\left(S, \gamma\right) + e\left(C, \delta\right) = 0$$

Batching equality checks

Let us denote the left-hand side of the above equality check by T. So, to verify a single proof, we must check that $T = 0$ in $\mathbb{G}_T$.

Now suppose we are given a batch of B proofs that we are to verify. We will keep notation as before but subscripts 1 to B. To verify all of those proofs, we must check that $T_i = 0$ holds for every $1 \leq i \leq B$.

Given an element $c \in \mathbb{F}_r$, we instead consider whether $\sum_{i=1}^{B} c^{i-1} \cdot T_i = 0$ holds. Clearly, if $T_i = 0$ holds for every $1 \leq i \leq B$, then $\sum_{i=1}^{B} c^{i-1} \cdot T_i = 0$ will hold as well. What about if $T_i = 0$ does not hold for every $1 \leq i \leq B$, how likely is it then that $\sum_{i=1}^{B} c^{i-1} \cdot T_i = 0$ still holds? We claim that this is very unlikely if c is a uniformly random element.

To show this, note that there is an isomorphism $f : \mathbb{G}_T \to \mathbb{F}_r$ as $\mathbb{F}_r$-modules. Thus, $\sum_{i=1}^{B} c^{i-1} \cdot T_i = 0$ is equivalent to $\sum_{i=1}^{B} f\left(T_i\right) \cdot f(c)^{i-1} = 0$. This can be rephrased as $f(c)$ being a root of the polynomial $\sum_{i=1}^{B} f\left(T_i\right) \cdot x^{i-1} \in \mathbb{F}_r[x]$. This polynomial is not zero by assumption (as at least one of the coefficients is nonzero) and has maximum degree B. There can thus be at most $B - 1$ many distinct roots, and hence, as c is a uniformly random element of $\mathbb{F}_r$, the chance that c is a root is at most $\frac{B-1}{r}$.

The above shows that as long as $B - 1$ is significantly smaller than r and c is chosen uniformly random, with the values T_i not depending on c, then the equality $\sum_{i=1}^{B} c^{i-1} \cdot T_i = 0$ is strong evidence for $T_i = 0$ holding for every $1 \leq i \leq B$.

It is important to stress how essential it is that the values T_i do not depend on c. If c were chosen first, but an attacker can still change T_i, then they could, for example, fix $T_2, \ldots, T_B$ to whatever value they want and then set $T_1 = -\left(\sum_{i=2}^{B} c^{i-1} \cdot T_i\right)$, which would make $T_1, \ldots, T_B$ pass the check.

We can summarize the discussion so far as follows. We consider the following protocol to verify a batch of proofs:

1. We are sent the verification keys, public inputs, and proofs.
2. We choose a uniformly random c.
3. We accept if $\sum_{i=1}^{B} c^{i-1} \cdot T_i = 0$ and reject otherwise.

What we argued for is that acceptance in step 3 is very unlikely to happen unless all individual proofs were valid.

In practice, such as in the universal batch verifier circuit, it may not be possible to make a uniformly random choice, but c must be deterministically derived somehow. In such cases, we might use a hash function H, which we model as a random oracle. The function H produces output in $\mathbb{F}_r$ that is uniformly random for each input, independently from the value at other inputs. However, evaluating H again on the same input will produce the same output again.

We can then use this modified protocol to verify a batch of proofs:

1. We are sent the verification keys, public inputs, and proofs.
2. We calculate T_i from the values received, and then $c = H\left(T_1, \ldots, T_B\right)$.
3. We accept if $\sum_{i=1}^{B} c^{i-1} \cdot T_i = 0$ and reject otherwise.

For any fixed values sent in step 1, the situation is as before: c is uniformly random, and furthermore the dependence on $T_1, \ldots, T_B$ means that the polynomial cannot be changed without changing the challenge point to a different uniformly random value. Thus, it is very unlikely that we will accept unless all proofs were valid.

In contrast to the previous variant, however, c is now predictable by the attacker. They can thus try locally whether their incorrect proofs will be rejected and try again and again before submitting. In this variant, there is thus the option to brute force locally (i.e., the attacker can keep trying different invalid proofs and check whether verification would succeed, until they find an invalid batch that nevertheless succeeds). If each verification is computationally intensive enough in combination with $\frac{B}{r}$ being large enough, finding an invalid batch like that would still be infeasible for the attacker, though.

Finally, it is not necessary for c to be obtained by applying H to $T_1, \ldots, T_B$. We can instead obtain c by applying H to values from which $T_1, \ldots, T_B$ are deterministically computed. Then it will still not be possible to change T_i without the challenge point changing to a new random value as well.

The batched check in the universal batch verifier chip

The universal batch verifier chip followed the above recipe, using the following for c.

$$c = H\left(||_{i=1}^{B}\left(H\left(vk_i\right)||\pi_i||P_i\right)\right)$$

The verification is then done split up into steps (the numbering of the steps follows the specification). In step 4, the values $S_i = s_{0,i} + \sum_{j=1}^{L} P_{j,i} \cdot s_{j,i}$ are calculated. This is implemented by the function `compute_pi_pairs`.

In step 5, implemented in `compute_pairs` (after a call to `compute_pi_pairs` to handle the previous step), the inputs to the pairing are scaled, calculating $\widetilde{A}_i = -c^{i-1}A_i$, $\widetilde{C}_i = c^{i-1}C_i$, $\widetilde{S}_i = c^{i-1}S_i$, and $\widetilde{\alpha}_i = c^{i-1}\alpha_i$.

In step 6, the multipairing is calculated to obtain the following value.

$$\sum_{i=1}^{B}\left(e\left(\widetilde{A}_i, B_i\right) + e\left(\widetilde{\alpha}_i, \beta_i\right) + e\left(\widetilde{S}_i, \gamma_i\right) + e\left(\widetilde{C}_i, \delta_i\right)\right)$$

The function `verify_with_challenge` is the one that calls the `multi_pairing` of the batch verifier chip in order to calculate this from the list of group elements obtained before. It then calls `check_pairing_result` to check that the resulting value is 0 (in our additive notation, in the code it is 1 due to use of multiplicative notation).

Let us assume the following.

- Assumption 1: $A_i, \alpha_i, s_{j,i}, C_i$ are elements of $\mathbb{G}_1$, and $B_i, \beta_i, \gamma_i,' \delta_i$ are elements of $\mathbb{G}_2$, and P_{j_i} are elements of $\mathbb{F}_r$ for all i and j.
- Assumption 2: The functions mentioned above implement the mathematical description mentioned above.

Then this is a secure way to perform a batch check, in the way discussed in the previous subsection. Indeed, we can expand the value that is checked against 0 as follows:

$$\begin{aligned}
&\sum_{i=1}^{B} \left(e\left(\widetilde{A}_i, B_i\right) + e\left(\widetilde{\alpha}_i, \beta_i\right) + e\left(\widetilde{S}_i, \gamma_i\right) + e\left(\widetilde{C}_i, \delta_i\right)\right) \\
=&\sum_{i=1}^{B} \left(e\left(-c^{i-1} \cdot A_i, B_i\right) + e\left(c^{i-1} \cdot \alpha_i, \beta_i\right) + e\left(c^{i-1} \cdot S_i, \gamma_i\right) + e\left(c^{i-1} \cdot C_i, \delta_i\right)\right) \\
=&\sum_{i=1}^{B} c^{i-1} \cdot \left(e\left(-A_i, B_i\right) + e\left(\alpha_i, \beta_i\right) + e\left(S_i, \gamma_i\right) + e\left(C_i, \delta_i\right)\right) \\
=&\sum_{i=1}^{B} c^{i-1} T_i
\end{aligned}$$

Furthermore, all values that T_i depend on are hashed to obtain c.

Relevant assumptions made by halo2-ecc's EccChip

Let us go through the functions of the external (and out of scope) halo2-ecc crate's ↗ EccChip and list the assumptions they make for correct functioning.

The function calls to halo2-ecc functions or chips by the UniversalBatchVerifierChip that we are going to analyze are the following:

- Call of `variable_base_msm` by `compute_pi_pairs` in order to calculate a partial sum for S_i (part of step 4).
- Call of `sum` by `compute_pi_pairs` in order to calculate S_i from the previous partial sum and an additional summand (part of step 4).
- Indirect call of `scalar_multiply` by `compute_pairs` in order to scale the inputs to the pairing with powers of the challenge (step 5).
- Indirect usage of the halo2-ecc PairingChip by `verify_with_challenge`.

Calculation of the right summand for S_i

For calculation of $S_i = s_{0,i} + \sum_{j=1}^{L} P_{j,i} \cdot s_{j,i}$, the relevant code in the `compute_pi_pairs` is the following:

```
let rhs = ec_chip.variable_base_msm::<G1Affine>(
    builder,
    &ss[1..], // We skip the first element
    inputs,
    F::NUM_BITS as usize,
);
let pi_term = ec_chip.sum::<G1Affine>(
    builder.main(0),
    once(rhs).chain(once(ss[0].clone())),
);
```

Let us first focus on the use of `variable_base_msm` to calculate $rhs = \sum_{j=1}^{L} P_{j,i} \cdot s_{j,i}$. Here, `ss[1..]` is $[s_{1,i}, \ldots, s_{L,i}]$ and $[P_{1,i}, \ldots, P_{L,i}]$. The function eventually calls a function documented as having these assumptions:

```
/// # Assumptions
/// * `points.len() == scalars.len()`
/// * `scalars[i].len() == scalars[j].len()` for all `i, j`
/// * `scalars[i]` is less than the order of `P`
/// * `scalars[i][j] < 2^{max_bits} for all j`
/// * `max_bits <= modulus::<F>.bits()`, and equality only allowed when the
    order of `P` equals the modulus of `F`
/// * `points` are all on the curve or the point at infinity
/// * `points[i]` is allowed to be (0, 0) to represent the point at infinity
    (identity point)
/// * Currently implementation assumes that the only point on curve with
    y-coordinate equal to `0` is identity point
```

In the relevant case, `max_bits == modulus::<F>.bits()`. The fifth point would then seem to disallow the points from not having the full order (so ruling out the point at infinity). It appears plausible that this is an inaccuracy in the documentation here, and what is required is only that $r \cdot P = 0$ for all input points P, which would also be satisfied by zero (the point at infinity).

What is more relevant is the requirement that all points lie on the curve or are the point at infinity. This assumption is not enforced by the UniversalBatchVerifierChip. An attacker can thus use arbitrary pairs for $s_{i,j}$ that may not satisfy the curve equation.

Calculation of S_i

Then, the EccChip's `sum` function is used to calculate the sum of $s_{0,i}$ with the previously calculated value. It makes the assumption that both summands are valid points on the curve and not the point at infinity. As mentioned before, $s_{i,j}$ are not constrained and choosable by the attacker, so this assumption is not necessarily satisfied for the first summand. As the second summand is the return value of `variable_base_msm`, and as just discussed, the assumptions for its argument may also be broken, the second summand may also not satisfy the assumptions made by `sum`. Furthermore, the second summand might legitimately be the point at infinity, which is also not allowed as input for `sum`. If the attacker uses invalid values for $s_{i,j}$, as discussed, or causes the second summand to be the point at infinity, we can then not rely on S_i to be a well-formed point on the curve either.

Scaling

The calculation of $\widetilde{A}_i = -c^{i-1}A_i$, $\widetilde{C}_i = c^{i-1}C_i$, $\widetilde{S}_i = c^{i-1}S_i$, and $\widetilde{\alpha}_i = c^{i-1}\alpha_i$ is done using a call to the EccChip's `scalar_multiply` function:

```
scalar_multiply::<_, FpChip<F, Fq>, G1Affine>(
    self.fp_chip,
    ctx,
    g1.clone(),
    vec![*scalar],
    F::NUM_BITS as usize,
    WINDOW_BITS,
)
```

That function makes the following assumptions:

```
/// # Assumptions
/// - `window_bits != 0`
/// - The order of `P` is at least `2^{window_bits}` (in particular, `P` is not
    the point at infinity)
/// - The curve has no points of order 2.
/// - `scalar_i < 2^{max_bits} for all i`
/// - `max_bits <= modulus::<F>.bits()`, and equality only allowed when the
    order of `P` equals the modulus of `F`
```

This is similar to the two previous functions. Here, again, `P` must be a valid point on the curve, and the point at infinity is not allowed. Now the proof points A_i and C_i are checked to be on the curve and not the point at infinity. However, α_i may be chosen arbitrarily by the attacker, and S_i is the result of operations whose inputs may not satisfy the required assumptions discussed above. Thus, we cannot rely on $\widetilde{\alpha}_i$ and $\widetilde{S}_i$ to be well-formed points on the curve.

Multipairing

The halo2-ecc PairingChip is then used to calculate the multipairing $\sum_{i=1}^{B}\left(e\left(\widetilde{A}_i, B_i\right) + e\left(\widetilde{\alpha}_i, \beta_i\right) + e\left(\widetilde{S}_i, \gamma_i\right) + e\left(\widetilde{C}_i, \delta_i\right)\right)$. This chip is somewhat less documented, but the comments for `miller_loop_BN` suggest that it is an assumption that the points are on the curve and that the points on the $\mathbb{G}_2$ side are not the zero point. Again, an attacker can freely choose β_i, γ_i and δ_i, breaking those assumptions. Furthermore, as discussed in the previous points, $\widetilde{\alpha}_i$ and $\widetilde{S}_i$ might not be well-formed points on the curve should the attacker have chosen α_i and $s_{j,i}$ to not be well-formed.

The upshot is that if the attacker manipulates verification keys in the batch, then the verification result deviates from its normal specification/interpretation. The previously discussed security proof for why the batched verification implies validity of the individual proofs thus does not apply anymore.

Impact assuming the check only depends on verification key, public inputs, and proofs

Let us for a moment assume that the above functions of the EccChip and PairingChip constrain their return values deterministically from their arguments even if the arguments do not satisfy the expected assumptions.

If the batch consisted of only a single proof, then the above suggests that it may be possible to construct a proof for the UniversalBatchVerifierChip circuit that accepts an incorrect proof for a malformed verification key. This is unlikely to be useful to the attacker, as useful attacks likely require being able to convince another party of the validity of a proof for a given circuit, for which the attacker can then not change the verification key.

The possibility to consider is thus whether an attacker would be able to construct a batch of, for example, two proofs that get accepted by the UniversalBatchVerifierChip circuit — with one proof being an incorrect proof for a legitimate verification key and the second proof using a malformed verification key that the attacker chose specifically so that the batch would be accepted. In such a case, the input pairs to the multipairing corresponding to the legitimate proof would be as intended. In pseudocode, the multipairing would then compute something like

```
out_of_spec_multipairing(legitimate_pairs,
    out_of_spec_scalar_multiplication_on_left_side(c,
    malformed_attacker_controlled_pairs))
```

or

```
out_of_spec_multipairing(malformed_attacker_controlled_pairs,
    scalar_multiplication_on_left_side(c, legitimate_pairs))
```

As we are assuming at the moment that these attacker-controlled pairs depend only on the verification key, public inputs, and proof, the attacker cannot change them without caus-

ing `c` to change as well. While we cannot reason about `out_of_spec_multipairing` and `out_of_spec_scalar_multiplication_on_left_side` without unpacking the implementation, it seems plausible that it might be infeasible in practice for an attacker to construct a malformed verification key making such a batch pass, even though the proof for the well-formed verification key is invalid. However, as this cannot be relied upon without deep analysis of the implementations in halo2-ecc, it should still be ensured that all assumptions made by the functions from halo2-ecc used are satisfied.

Nondeterminism

In the previous section, we briefly discussed the impact, assuming that the used functions of the EccChip and PairingChip constrain their return values deterministically from their arguments even if the arguments do not satisfy the expected assumptions. However, this assumption is actually not true, as we will now see.

The `sum` function

Let us begin by looking into the implementation of `sum` in the EccChip of halo2-ecc:

```
/// None of elements in `points` can be point at infinity.
pub fn sum<C>(
    &self,
    ctx: &mut Context<F>,
    points: impl IntoIterator<Item = EcPoint<F, FC::FieldPoint>>,
) -> EcPoint<F, FC::FieldPoint>
where
    C: CurveAffineExt<Base = FC::FieldType>,
{
    let rand_point = self.load_random_point::<C>(ctx);
    let rand_point = into_strict_point(self.field_chip, ctx, rand_point);
    let mut acc = rand_point.clone();
    for point in points {
        let _acc = self.add_unequal(ctx, acc, point, true);
        acc = into_strict_point(self.field_chip, ctx, _acc);
    }
    self.sub_unequal(ctx, acc, rand_point, true)
}
```

Note that instead of directly computing the sum, this function starts out with a (intended random) point that is subtracted again at the end. The reason for this is that the general formulas for elliptic curve addition with points in affine coordinates do not apply if the two points are equal or additive inverses of each other. To be able to use only a single formula for addition, a trick is thus used: if R is a random point, then it is unlikely that R would be equal to $\pm P$ or that $R + P$ would be equal to $\pm Q$, so by performing the calculation as $((R + P) + Q) - R$, most special cases can be avoided. The

one special case that is not avoidable is $P = -Q$, as then one would have $(R + P) + Q = R$. See Finding 3.2. ↗.

The random point is loaded by the following function:

```
pub fn load_random_point<F, FC, C>(chip: &FC, ctx: &mut Context<F>) ->
    EcPoint<F, FC::FieldPoint>
where
    F: PrimeField,
    FC: FieldChip<F>,
    C: CurveAffineExt<Base = FC::FieldType>,
{
    let base_point:
    C = C::CurveExt::random(ChaCha20Rng::from_entropy()).to_affine();
    let (x, y) = base_point.into_coordinates();
    let base = {
        let x_overflow = chip.load_private(ctx, x);
        let y_overflow = chip.load_private(ctx, y);
        EcPoint::new(x_overflow, y_overflow)
    };
    // for above reason we still need to constrain that the witness is on the
    curve
    check_is_on_curve::<F, FC, C>(chip, ctx, &base);
    base
}
```

As can be seen here, on witness generation, the point is generated randomly. An attacker must not follow this, however, and can choose specific values. The only in-circuit constraint imposed here is that the point is actually on the curve. The `check_is_on_curve` function checks that the pair of coordinates satisfies the curve equation, so the point at infinity is ruled out here.

The implementations of `ec_add_unequal` and `ec_sub_unequal` (which are the functions ultimately called by `add_unequal` and `sub_unequal`) do not treat $(0, 0)$, which is used to represent the point at infinity, in a separate way, as can be seen here in the example of addition:

```
// Implements:
// Given P = (x_1, y_1) and Q = (x_2, y_2), ecc points over the field F_p
// assume x_1 != x_2
// Find ec addition P + Q = (x_3, y_3)
// By solving:
// lambda = (y_2-y_1)/(x_2-x_1) using constraint
// lambda * (x_2 - x_1) = y_2 - y_1
// x_3 = lambda^2 - x_1 - x_2 (mod p)
// y_3 = lambda (x_1 - x_3) - y_1 mod p
//
/// If `is_strict = true`, then this function constrains that `P.x != Q.x`.
```

```
/// If you are calling this with `is_strict = false`, you must ensure that `P.x
    != Q.x` by some external logic (such
/// as a mathematical theorem).
///
/// # Assumptions
/// * Neither `P` nor `Q` is the point at infinity (undefined behavior
    otherwise)
pub fn ec_add_unequal<F: PrimeField, FC: FieldChip<F>>(
    chip: &FC,
    ctx: &mut Context<F>,
    P: impl Into<ComparableEcPoint<F, FC>>,
    Q: impl Into<ComparableEcPoint<F, FC>>,
    is_strict: bool,
) -> EcPoint<F, FC::FieldPoint> {
    let (P, Q) = check_points_are_unequal(chip, ctx, P, Q, is_strict);

    let dx = chip.sub_no_carry(ctx, &Q.x, &P.x);
    let dy = chip.sub_no_carry(ctx, Q.y, &P.y);
    let lambda = chip.divide_unsafe(ctx, dy, dx);

    // x_3 = lambda^2 - x_1 - x_2 (mod p)
    let lambda_sq = chip.mul_no_carry(ctx, &lambda, &lambda);
    let lambda_sq_minus_px = chip.sub_no_carry(ctx, lambda_sq, &P.x);
    let x_3_no_carry = chip.sub_no_carry(ctx, lambda_sq_minus_px, Q.x);
    let x_3 = chip.carry_mod(ctx, x_3_no_carry);

    // y_3 = lambda (x_1 - x_3) - y_1 mod p
    let dx_13 = chip.sub_no_carry(ctx, P.x, &x_3);
    let lambda_dx_13 = chip.mul_no_carry(ctx, lambda, dx_13);
    let y_3_no_carry = chip.sub_no_carry(ctx, lambda_dx_13, P.y);
    let y_3 = chip.carry_mod(ctx, y_3_no_carry);

    EcPoint::new(x_3, y_3)
}
```

If at least one of the summands is not a valid affine point on the curve, `ec_add_unequal` will return an affine point deterministically calculated from the input, but without interpretation in terms of addition on the elliptic curve (similarly for `ec_sub_unequal`).

As discussed before, in the calculation of S_i, the above `sum` function is used and an attacker can make one or both of the summands the point at infinity, which is represented by $(0, 0)$. The important thing to note now is that the incorrect result returned by `ec_add_unequal` and `ec_sub_unequal` for invalid arguments do not behave in the usual way such that the random point R cancels out. This means that calling `sum` with one of the summands being $(0, 0)$ will return a result that is dependent on the random point R being loaded by `sum` itself.

This dependence can be verified by the following SageMath script:

```
### bn254
p = 0x30644e72e131a029b85045b68181585d97816a916871ca8d3c208c16d87cfd47
K = GF(p)
a = K(0x0000000000000000000000000000000000000000000000000000000000000000)
b = K(0x0000000000000000000000000000000000000000000000000000000000000003)
E = EllipticCurve(K, (a, b))
r = 0x30644e72e131a029b85045b68181585d2833e84879b9709143e1f593f0000001
#G = E(0x2523648240000001BA344D8000000086121000000000013A700000000000012,
    0x0000000000000000000000000000000000000000000000000000000000000001)
#E.set_order(0x2523648240000001BA344D8000000007FF9F800000000010A10000000000000D
    * 0x01)


# The following two functions reflect the implementation in the halo2-ecc
# EccChip.

# ec_add_unequal
def add(P, Q):
  (x_1, y_1) = P
  (x_2, y_2) = Q
  assert x_1 != x_2
  lam = (y_2-y_1)/(x_2-x_1)
  x_3 = lam^2 - x_1 - x_2
  y_3 = lam*(x_1 - x_3) - y_1
  return (x_3, y_3)

# ec_sub_unequal
def sub(P, Q):
  (x_1, y_1) = P
  (x_2, y_2) = Q
  assert x_1 != x_2
  lam = -(y_2+y_1)/(x_2-x_1)
  x_3 = lam^2 - x_1 - x_2
  y_3 = lam*(x_1 - x_3) - y_1
  return (x_3, y_3)


# Sanity check: for random legitimate points on the curve, the above functions
# should be addition / subtraction
P = E.random_point()
Q = E.random_point()
assert ((P+Q).xy() == add(P.xy(), Q.xy()))
assert ((P-Q).xy() == sub(P.xy(), Q.xy()))

def is_on_curve(P):
  return P[1]**2 == P[0]**3 + 3
```

```
assert is_on_curve(E.random_point().xy())

# Tests whether f (that takes a random value as input) is independent of it,
# and whether the output is on the curve.
def test_constant_and_on_curve(f):
  R1, R2 = (E.random_point().xy() for _ in range(2))
  f1 = f(R1)
  f2 = f(R2)
  return (f1 == f2, is_on_curve(f1))

def sum_func(P, Q):
  def helper(R):
    return sub(add(add(R, P), Q), R)
  return helper

def test_left_zero():
  P = (K(0),K(0))
  Q = E.random_point().xy()
  return test_constant_and_on_curve(sum_func(P, Q))

def test_right_zero():
  P = E.random_point().xy()
  Q = (K(0),K(0))
  return test_constant_and_on_curve(sum_func(P, Q))

def test_none_zero():
  P = E.random_point().xy()
  Q = E.random_point().xy()
  return test_constant_and_on_curve(sum_func(P, Q))

# If both points are valid, then sum does not depend on the random point and
# the output is on the curve
assert test_none_zero() == (True, True)
# but if one of the two summands is (0,0), then the output does depend on the
# random point, and the output may not be on the curve
assert test_left_zero() == (False, False)
assert test_right_zero() == (False, False)
```

The `variable_base_msm` and `scalar_multiply` functions

The `variable_base_msm` and `scalar_multiply` functions are ultimately implemented by either by `multi_scalar_multiply` or `pippenger::multi_exp_par`. Both use a random-point trick analogous to what was discussed for `sum`. In contrast to `sum`, both functions are intended to deal correctly with $(0, 0)$ as representing the point at infinity, according to the documentation. While we did not test this, it appears highly likely that the output will be dependent on the random point, however, should

one or more of the input points fail to satisfy the curve equation while being unequal to $(0, 0)$.

Impact of nondeterminism

The dependence of the result of the multipairing on these points R in the cases just discussed means that an attacker still has parameters they can change *after* the challenge c has been fixed. The case of a malformed verification key could be dealt with by rejecting a verified batch out-of-circuit whenever one of the included verification keys is malformed. However, as discussed before, the attacker can also introduce dependence on such a point R by setting all public inputs to zero. For each proof in the batch for which the attacker does this, they essentially gain one degree of freedom to influence the result of the multipairing. Without unpacking in detail the implementation of the relevant functions, including the multi-Miller loop, it is difficult to say how feasible it will be for an attacker to make choices for these points R with which they can make the universal batch verifier circuit accept a batch that includes an invalid proof for a legitimate (not attacker chosen) verification key. However, it appears plausible that this could be possible.

Recommendations

All points in the verification key should be checked to actually lie on the curve. As the universal batch verifier circuit exposes these points as public instances, this does not necessary need to be done in circuit, but in that case, it should be documented clearly that this must be checked outside.

Furthermore, the possibility that `sum` might be called in the `compute_pi_pairs` function with the second summand being the point at infinity should be removed.

One way to do that would be to use `variable_base_msm` with points `ss` and scalars `[1] + input` to calculate the entire sum $S_i = s_{0,i} + \sum_{j=1}^{L} P_{j,i} \cdot s_{j,i}$ directly, rather than calling it with only points `ss[1..]` and scalars `input` and then using `sum` to add `ss[0]` to this intermediate result.

Remediation

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

- `e19c58f8` ↗
- `a9b03a64` ↗
- `b6d39db7` ↗

3.2. Proving fails for public inputs that are all zero

Target	UniversalBatchVerifierChip		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

The universal batch verifier chip's `compute_pi_pairs` function is used to calculate a sum $S = s_0 + \sum_{j=1}^{L} P_j \cdot s_j$ using the following code, where `ss` corresponds to $[s_0, \ldots, s_L]$ and `inputs` to $[P_1, \ldots, P_L]$. The elliptic curve points $s_0, \ldots, s_L$ are part of the verification key for a Groth16 circuit, and the scalars $P_1, \ldots, P_L$ are the public inputs.

```
let rhs = ec_chip.variable_base_msm::<G1Affine>(
    builder,
    &ss[1..], // We skip the first element
    inputs,
    F::NUM_BITS as usize,
);
let pi_term = ec_chip.sum::<G1Affine>(
    builder.main(0),
    once(rhs).chain(once(ss[0].clone())),
);
```

If all components of $[P_1, \ldots, P_L]$ are zero, then `ec_chip.variable_base_msm` will return zero (i.e., the point at infinity), represented by $(0, 0)$. There are also other linear combinations of $s_1, \ldots, s_L$ that are zero, though this is unlikely to happen by accident for legitimate verification keys. If $s_1, \ldots, s_L$ were random points, then finding nonzero scalars that make the linear combination zero means solving a discrete log relation problem for the elliptic curve, which is assumed to be infeasible. Note that if it were feasible to find a nontrivial linear relation between the $s_1, \ldots, s_L$, then this would amount to malleability of Groth16 proofs with respect to the public inputs. Thus, the case to consider here is the one where all scalars are zero.

Whenever the linear combination `rhs` is zero, the first summand in the call to `ec_chip.sum` will be $(0, 0)$. However, `sum` does not support this as a representation of the point at infinity, so the result `pi_term` will not be `ss[0]` as it should. For more details, see Finding 3.1. ↗.

Similarly, proving will fail should `rhs` and `ss[0]` be additive inverses of each other (equivalently, if the correct result for `pi_term` would be zero). This is again unlikely to happen by accident for legitimate verification keys. For more details, see Finding 3.1. ↗.

Impact

Attempting to prove verification of a batch of Groth16 proofs using the universal batch verifier chip will fail if one of the proofs in the batch has all public inputs being zero. Proof generation will also fail in some other cases that are unlikely to occur for legitimate verification keys but can be caused by specifically choosing the verification key. The concern in this latter case is not completeness but soundness, and relevant impacts are discussed in Finding 3.1. ↗.

Recommendations

We recommend to change the above snippet of code to one that correctly computes the sum $S = s_0 + \sum_{j=1}^{L} P_j \cdot s_j$ in all cases in which P_j are points on the curve. This could be done by using `variable_base_msm` for the entire sum, by replacing `ss[1..]` in the call by `ss` and `inputs` by the concatenation of `F::one()` with `inputs`.

Remediation

This issue has been acknowledged by Nebra, and a fix was implemented in commit `e19c58f8` ↗.

3.3. Wrong bitmask-length computation

Target	Keccak and UniversalBatchVerifierChip		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

Two almost identical functions named `first_i_bits_bitmask` are used to create a bitmask with `b[j]=1` when `i < j`. Those functions are used to verify variable-length Keccak padding of bytes into 64-bit words and to check the padding of public inputs in the universal batch verifier.

The size of the bitmask n is given as a parameter of the function. The bitmask is built by pushing constraints computed by the function `is_less_than` from the Axiom implementation ↗:

```
pub fn first_i_bits_bitmask<F: ScalarField>(
    ctx: &mut Context<F>,
    chip: &impl RangeInstructions<F>,
    i: AssignedValue<F>,
    n: u64,
) -> Vec<AssignedValue<F>> {
    let mut bitmask = Vec::with_capacity(n as usize);
    let num_bits = log2(n as usize);
    for idx in 0..n {
        let idx = ctx.load_constant(F::from(idx));
        bitmask.push(chip.is_less_than(ctx, idx, i, num_bits as usize));
    }
    bitmask
}
```

The `is_less_than` function takes as a parameter `num_bits`, which is assumed to be the number of bits to represent the values compared. In the `first_i_bits_bitmask` functions, if the size of the mask is a power of two, let us say 2^m, then the number of bits passed to the `is_less_than` function is computed with the `log2` function, which in the previous example will output m, but the compared values require $m + 1$ bits to be represented. It contradicts the assumption of the `is_less_than` function on `num_bits`.

Impact

It seems that the code of `is_less_than` is currently able to handle properly such edge cases, but nothing guarantees that future updates will not break such behavior. If so, the constraints may not be sound anymore for lengths that equal a power of two.

Recommendations

Even if it appears so far that the chip handles this edge case correctly, it may be safer to compute the number of bits with the following:

```
let num_bits = log2((n + 1) as usize);
```

Remediation

This issue has been acknowledged by Nebra, and a fix was implemented in commit 48ca21a3 ↗.

3.4. Method `instances` of `AssignedBatchEntries` does not follow specification

Target	AssignedBatchEntries		
Category	Code Maturity	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

In `circuits/src/batch_verify/universal/chip.rs`, the struct `AssignedBatchEntries` has the following method implemented:

```
/// Returns the instances, consuming `self`.
pub fn instance(self) -> Vec<AssignedValue<F>> {
    self.0
        .into_iter()
        .flat_map(|entry| {
            once(entry.len).chain(entry.public_inputs.into_iter())
        })
        .collect()
}
```

This only returns the lengths of the public inputs and the public inputs themselves. From the description and specification for the universal batch verifier, it would be expected that circuit IDs are also returned.

Impact

In the reviewed in-scope codebase, this method is unused, so there is no impact. Deviation from the specification could cause mistakes after future code changes, however.

Recommendations

We recommend to change the implementation to return the public instances of the universal batch verifier as specified, or alternatively remove this function.

Remediation

This issue has been acknowledged by Nebra, and a fix was implemented in commit `59df2dc3` ↗.

3.5. Undocumented assumption for `select_with_bitmask`

Target	hashing.rs		
Category	Code Maturity	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

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

```
/// Selects the right `elements` using `bitmask`.
///
/// # Note
///
/// The elements in `bitmask` must be constrained to be booleans.
fn select_with_bitmask<F: EccPrimeField>(
    ctx: &mut Context<F>,
    chip: &GateChip<F>,
    bitmask: &[AssignedValue<F>],
    elements: Vec<Vec<AssignedValue<F>>>,
) -> Vec<AssignedValue<F>> {
    assert_eq!(
        bitmask.len(),
        elements.len(),
        "Bitmask and elements length mismatch"
    );
    let inner_len = elements
        .first()
        .expect("elements must have at least one element")
        .len();
    let mut result = Vec::with_capacity(inner_len);
    for index in 0..inner_len {
        let elmts = elements.iter().map(|inner_vec| inner_vec[index]);
        let bits = bitmask.iter().map(|b| QuantumCell::<F>::from(*b));
        result.push(chip.inner_product(ctx, elmts, bits))
    }
    result
}
```

This function is intended to select the inner list from the list of lists `elements` that is specified via the

bitmask `bitmask`. So for example, if `bitmask = [0, 1]` and `elements = [[x,y,z], [a,b,c]]`, then it would return a list of circuit variables constrained to be equal to `[a,b,c]`. The description so far would also make sense if the inner lists had different lengths. For example, it would be reasonable that for `bitmask = [0, 1]` and `elements = [[x], [a,b,c]]`, the return value should still be `[a,b,c]`. However, as the implementation obtains the length of the list that is returned from the length of the first inner list, only `[a]` would be returned.

Impact

The current in-scope codebase does not appear to call `select_with_bitmask` with arguments `elements` where the lengths of the inner lists differ. However, future coding mistakes in seemingly unrelated parts of the code could cause issues. For example, if `elements` were prepended by an empty list by accident, then `select_with_bitmask` would always return an empty list without erroring, which could lead to, for example, not hashing elements that were intended to be hashed.

Recommendations

We recommend to document that this function assumes all inner lists are of the same length and to assert this in the implementation as well to prevent accidental incorrect uses in the future.

Remediation

This issue has been acknowledged by Nebra, and a fix was implemented in commit `59df2dc3` ↗.

3.6. Poseidon edge cases

Target	hashing.rs		
Category	Code Maturity	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

In `circuits/src/utils/hashing.rs`, the function `var_len_poseidon_no_len_check` makes some assumptions that are not fully documented or could be checked in code to prevent incorrect usage.

The following is an extract of the function's implementation:

```
/// Computes the poseidon hash of `assigned[..num_parts]`.
///
/// # Note
///
/// `num_parts` must be constrained to be smaller than `assigned.max_parts()`.
pub fn var_len_poseidon_no_len_check<F, T>(
    ctx: &mut Context<F>,
    domain_tag_str: Option<&str>,
    chip: &GateChip<F>,
    num_parts: AssignedValue<F>,
    assigned: &T,
) -> AssignedValue<F>
where
    F: EccPrimeField,
    T: InCircuitPartialHash<F>,
{
    let max_parts = assigned.max_parts();
    // `PoseidonHasher` absorbs the field elements in `POSEIDON_R`-sized
    chunks.
    // For possible number of parts, we compute how many chunks
    `PoseidonHasher` would
    // be absorbing (more precisely, the corresponding chunk indices), as well
    as the
    // remainder, i.e., how many extra field elements are there which wouldn't
    fill a chunk.
    let number_of_elements = (0..max_parts)
        .map(|parts| {
            assigned.parts_to_num_elements(parts)
                + (domain_tag_str.is_some() as usize)
```

```
        })
        .collect_vec();
    let (chunk_indices, remainder_lens): (Vec<_>, Vec<_>) = number_of_elements
        .iter()
        .map(|num_elts| (num_elts / POSEIDON_R, num_elts % POSEIDON_R))
        .unzip();

    // ...

     // Retrieve all intermediate states (after the absorption of each chunk).
    let (intermediate_states, is_remainder_len_zero)
    = hasher.poseidon.intermediate_states(ctx, chip);

    // ...

    // Take only the states with the relevant chunk indexes. Note that these
    // indices are given by type T's implementation of `InCircuitPartialHash`,
    // so they don't need to be selected in-circuit: they are constant and
    don't
    // depend on `num_parts`.
    let intermediate_states = intermediate_states
        .into_iter()
        .enumerate()
        .filter(|(index, _)| chunk_indices.contains(index))
        .map(|(_, state)| state.to_vec())
        .collect_vec();

    // Compute the `num_parts`-th bit bitmask.
    let bitmask = ith_bit_bitmask(ctx, chip, num_parts, max_parts as u64);
    // Select the right intermediate state
    let state = select_with_bitmask(ctx, chip, &bitmask, intermediate_states);

    // ...
}
```

We can see that `number_of_elements` is a list where the component with index `i` contains the number of field elements that the first `i` parts of `assigned` would use, possibly together with the domain tag if relevant. Then, `chunk_indices` is calculated as the list that as component with index `i` contains the number of full chunks that would be filled by `number_of_elements[i]` many field elements. For example, it could be that each part consists of six field elements, and there is no domain tag. Then, with `POSEIDON_R = 2`, we would have `number_of_elements = [0, 6, 12, 18, ...]` and `chunk_indices = [0, 3, 6, 9, ...]`.

In the first assignment of `intermediate_states`, it will be a list whose component at index `i` is the state after absorbing the first `i` chunks. To be able to talk about the reassigned `intermediate_states`, we will denote it by `intermediate_states_new`. The intention is that `intermediate_states_new[i] = intermediate_states[chunk_indices[i]]`. So in our example above,

the intention is that `intermediate_states_new = [intermediate_states[0], intermediate_states[3], intermediate_states[6], intermediate_states[9], ...]`.

This is implemented by filtering `intermediate_states` and only keeping those components whose index appears in `chunk_indices`. This works for the example considered. However, it assumes that the components of `chunk_indices` do not repeat. If `InCircuitPartialHash` were implemented for a type for which each part takes no field elements to store (for example, the edge case of lists of tuples of length zero), then we would have `chunk_indices = [0, 0, 0, ...]`, and the intention would be to get `intermediate_states_new = [intermediate_states[0], intermediate_states[0], intermediate_states[0], ...]`. However, instead we will have `intermediate_states_new = [intermediate_states[0]]`, and the call to `select_with_bitmask` will panic because `bitmask` and `intermediate_states_new` will have different lengths.

Another assumption that is made by the implementation of `var_len_poseidon_no_len_check` that is not fully documented is that `assigned.max_parts()` and `assigned.parts_to_num_elements(parts)` need to be independent of values (so only depend on types).

Impact

The current in-scope codebase satisfies the assumptions needed for `var_len_poseidon_no_len_check` to work correctly. To prevent issues arising with future code changes, we nevertheless recommend documenting the assumptions made by the implementation.

Recommendations

We recommend to document that this function assumes that `assigned.max_parts()` and `assigned.parts_to_num_elements(parts)` need to be independent of values. Furthermore, we recommend to document that `assigned.parts_to_num_elements` must be injective (not have reoccurring return values) and assert in code that no components of `chunk_indices` reoccur. Alternatively, change the implementation of the snippet

```
let intermediate_states = intermediate_states
    .into_iter()
    .enumerate()
    .filter(|(index, _)| chunk_indices.contains(index))
    .map(|(_, state)| state.to_vec())
    .collect_vec();
```

to also cover the case of reoccurring chunk indexes by implementing this (pseudocode): `intermediate_states = [intermediate_states[chunk_indices[i]] for i in 0..max_parts]`.

Remediation

This issue has been acknowledged by Nebra, and a fix was implemented in commit 59df2dc3 ↗.

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. Documentation is sometimes misaligned with the implementation

During the review of the codebase, we noticed some places where behavior of the code deviates from the documented behavior.

For example, the NEBRA UPA v1.1.0 Protocol Specification refers in "Circuit Statements" to the Groth16 verification with an outdated link.

In `circuits/src/keccak/mod.rs`, there are several functions with comments where the order in which things are appended is opposite to the implementation — for example, the function `flatten`:

```
/// Appends the vk hash to the application public inputs.
pub fn flatten(&self) -> Vec<AssignedValue<F>> {
    let mut result = vec![self.app_vk_hash];
    result.extend_from_slice(&self.app_public_inputs);
    result
}
```

The result begins with the vk hash, so it would be correct to document this as "Appends the application public inputs to the vk hash" or "Concatenates the vk hash and the application public inputs".

The documenting comment for the function `num_field_elements` in the same file appears to be incorrect as well.

```
/// Returns the number of field elements which will be keccak'd together.
#[cfg(test)]
pub fn num_field_elements(&self) -> usize {
    self.len.value().get_lower_32() as usize
}
```

Here, the return value is the number of public inputs. However, the circuit ID is also part of the field elements that the Keccak hash is taken of. Thus, the implementation does not match the specification given by the preceding comment.

The above two examples were corrected in commit `59df2dc3` ↗

4.2. Additional asserts

In `circuits/src/batch_verify/common/chip.rs`, the function `scale_pairs` implemented for BatchVerifierChip is implemented as follows:

```
/// Returns `(scalar_i * A_i, B_i)`
pub(crate) fn scale_pairs(
    &self,
    ctx: &mut Context<F>,
    scalars: &[AssignedValue<F>],
    pairs: &[(G1Point<F>, G2Point<F>)],
) -> Vec<(G1Point<F>, G2Point<F>)> {
    let mut result = Vec::with_capacity(pairs.len());
    for ((g1, g2), scalar) in pairs.iter().zip(scalars.iter()) {
        result.push((
            scalar_multiply::<_, FpChip<F, Fq>, G1Affine>(
                self.fp_chip,
                ctx,
                g1.clone(),
                vec![*scalar],
                F::NUM_BITS as usize,
                WINDOW_BITS,
            ),
            g2.clone(),
        ))
    }
    result
}
```

If the lists `scalars` and `pairs` have different lengths, `scale_pairs` will return a list that is only as long as the shorter of the two lists but will not error. For in-depth defense against future coding errors in callers, we recommend to change usage of `zip` to `zip_eq` or otherwise verify that `scalars` and `pairs` have the same lengths.

Similarly, the `pack` function in `circuits/src/keccak/utils.rs` should not be used with an array `bits` that is longer than 64 entries. To prevent future mistakes, it could be asserted in the function that `bits` is at most 64 entries long:

```
fn pack<F>(
    ctx: &mut Context<F>,
    chip: &RangeChip<F>,
    bits: &[AssignedValue<F>],
) -> AssignedValue<F>
where
    F: EccPrimeField,
{
```

```
    assert_eq!(bits.len(), 64, "Number of bits to be packed should be 64");
    let base = ctx.load_constant(F::from(8u64));
    let initial_value = ctx.load_constant(F::zero());
    bits.iter().rev().fold(initial_value, |acc, bit| {
        chip.gate.mul_add(ctx, acc, base, *bit)
    })
}
```

The two asserts discussed above were inserted in commit 59df2dc3 ↗

4.3. Endianess mismatch between Keccak circuit specification and implementation

The function `compose_into_field_elements` in `circuits/src/keccak/utils.rs` is implemented as follows:

```
/// Composes `bytes` into two 16-byte field elements. Each field element in
    `bytes`
/// is assumed to have been previously range checked.
///
/// # Specification
///
/// This function performs **Step 5: compose into field elements**
/// in the variable length keccak spec.
pub fn compose_into_field_elements<F: EccPrimeField>(
    ctx: &mut Context<F>,
    chip: &RangeChip<F>,
    bytes: &[AssignedValue<F>; 32],
) -> [AssignedValue<F>; 2] {
    let byte_decomposition_powers = byte_decomposition_powers()
        .into_iter()
        .map(|power| QuantumCell::from(ctx.load_constant(power)))
        .collect_vec();
    let result = bytes
        .iter()
        .rev()
        .chunks(16)
        .into_iter()
        .map(|chunk| {
            chip.gate.inner_product(
                ctx,
                chunk.into_iter().cloned().map(QuantumCell::from),
                byte_decomposition_powers.clone(),
```

```
            )
        })
        .collect::<Vec<_>>();
    result
        .try_into()
        .expect("Conversion from vec into array is not allowed to fail")
}
```

The function `byte_decomposition_powers` in turn is implemented as follows:

```
/// Returns an iterator of field elements over the powers of 2^8.
pub(crate) fn byte_decomposition_powers<F: EccPrimeField>() -> Vec<F> {
    let num_bytes = num_bytes::<F>();
    let mut powers = Vec::<F>::with_capacity(num_bytes);
    let two_to_eight = F::from(1 << BYTE_SIZE_IN_BITS);

    powers.push(F::one());
    for i in 1..num_bytes {
        powers.push(powers[i - 1] * two_to_eight);
    }

    powers
}
```

As can be seen here, the return value of `byte_decomposition_powers` is the list of powers of 2^8 in increasing order — so starting with $1, 2^8, \left(2^8\right)^2, \ldots$.

Let us denote the components of `bytes` by $b_0, \ldots, b_{31}$. Then, the `bytes` array back in `compose_into_field_elements` is first reversed by `rev()`, resulting in an iterator over $[b_{31}, \ldots b_0]$. Chunking results in chunks $[b_{31}, \ldots b_{16}]$ and $[b_{15}, \ldots b_0]$. The mapped call of `inner_product` then takes the inner product of each chunk with $\left[1, 2^8, \left(2^8\right)^2, \ldots, \left(2^8\right)^1 5\right]$. The result will thus consist of $F_1 = \sum_{k=0}^{15} b_{16+15-k} \cdot 2^{8 \cdot k}$ and $F_2 = \sum_{k=0}^{15} b_{15-k} \cdot 2^{8 \cdot k}$. This does not match the specification for step 5 of the variable-length Keccak circuit, which (with slightly different notation) suggests that $F_1 = \sum_{k=1}^{16} b_k \cdot 2^{8 \cdot k}$ instead.

Apart from the sum in the specification starting with 1 instead of 0, the discrepancy can be summarized as the implementation treating the bytes as being given in big endian, whereas the specification suggests little endian.

The discrepancy discussed above was corrected in the specification in commit 5927809d ↗

4.4. Different assert than intended

The `assert_var_len_keccak_correctness` function in `circuits/src/keccak/chip.rs` makes some consistency checks of a couple of parameters. They are not in-circuit checks, so they are not directly security relevant but rather sanity checks to defend against coding mistakes elsewhere. The function is implemented as follows:

```
/// Checks that the chip has executed [`keccak_var_len`](Self::keccak_var_len)
/// correctly. This function should be called before updating the chip inner
    state.
fn assert_var_len_keccak_correctness(
    &self,
    output_bytes: &[u8],
    output_assigned: &[AssignedValue<F>],
    max_input_bytes_length_after_padding: usize,
    unpadded_input_bytes_length: usize,
    byte_len_val: usize,
    max_len: usize,
) {
    assert!(max_len < max_input_bytes_length_after_padding);
    assert!(max_input_bytes_length_after_padding >= byte_len_val);
    assert_eq!(byte_len_val % NUM_BYTES_PER_WORD, 0);
    assert_eq!(max_input_bytes_length_after_padding % RATE, 0);
    assert_eq!(
        output_bytes.to_vec(),
        get_assigned_bytes_values(output_assigned)
    );
    assert_eq!(
        unpadded_input_bytes_length,
        max((max_len / RATE) * RATE, byte_len_val)
    );
}
```

Here, `byte_len_val` is the original input length, `max_len` is the maximum input length, and `max_input_bytes_length_after_padding` is the maximum input length after padding has been applied. The first assert checks that `max_len < max_input_bytes_length_after_padding`, which should hold as padding always increases the length. While `max_input_bytes_length_after_padding >= byte_len_val` should also hold for the same reason, checking for greater-or-equal here would be weaker than needed; `max_input_bytes_length_after_padding` should always be strictly bigger than `byte_len_val`. It is likely what is actually intended here is to check `max_len >= byte_len_val`, which should hold but is not currently checked by the implementation. Replacing the check in the second assert with this inequality would also imply the currently asserted inequality, as one would then have `max_input_bytes_length_after_padding > max_len >= byte_len_val`.

The assert was corrected as suggested in commit `59df2dc3` ↗

4.5. Missing tests

As public inputs for proofs verified by the universal batch verifier are padded with zero, Finding 3.2. ↗ would have been caught by a test case where a proof in the batch has no public inputs. However, the code-sampling test cases did not cover this edge case (extract from `circuits/src/batch_verify/universal/types.rs`):

```
/// Samples a [`UniversalBatchVerifierInput`] for `config`.
pub fn sample<R>(config: &UniversalBatchVerifierConfig, rng: &mut R) -> Self
where
    R: RngCore + ?Sized,
{
    let num_public_inputs = config.max_num_public_inputs as usize;
    let length = rng.gen_range(1..=num_public_inputs);
    let (proofs_and_inputs, vk) = sample_proofs_inputs_vk(length, 1, rng);
    let (proof, inputs) = proofs_and_inputs[0].clone();

    assert!(length > 0);
    assert!(length + 1 == vk.s.len());
    assert!(length == inputs.0.len());

    Self { vk, proof, inputs }
}
```

As can be seen here, only cases with at least one public input are tested. We recommend to change `rng.gen_range(1..=num_public_inputs);` to `rng.gen_range(0..=num_public_inputs);` to also cover the case of no public inputs.

Testing only with randomly generated inputs generated as above may miss special edge-case values with low density such as a public input being zero (a random field element is extremely unlikely to be zero) or parameters occurring more than once. We thus recommend to also include test cases with such human-defined edge cases.

Testing for all public inputs being zero was added in commits e19c58f8 ↗, ad49aa7e ↗, and 5c2a1590 ↗.

5. Circuit descriptions

In this section we provide descriptions of some circuits or parts of circuits. As time permitted, we created a written explanation detailing how the circuit's constraints enforce the intended relations between the public circuit variables. To streamline the exposition, we may at times describe parts of the circuit as if findings detailed in section 3. ↗ were fixed as recommended.

5.1. Universal batch verifier circuit

The universal batch verifier circuit is specified in detail in `spec/circuits/universal_batch_verifier.md`. There is one discrepancy between the code and the specification that we pointed out in Discussion section 4.3. ↗ — for the purposes of the following description, we are going to assume that the endianess in the implementation is correct and assume a correspondingly fixed specification. We will also not mention the findings again here.

We will discuss how the specification corresponds to the implementation, starting from the highest level.

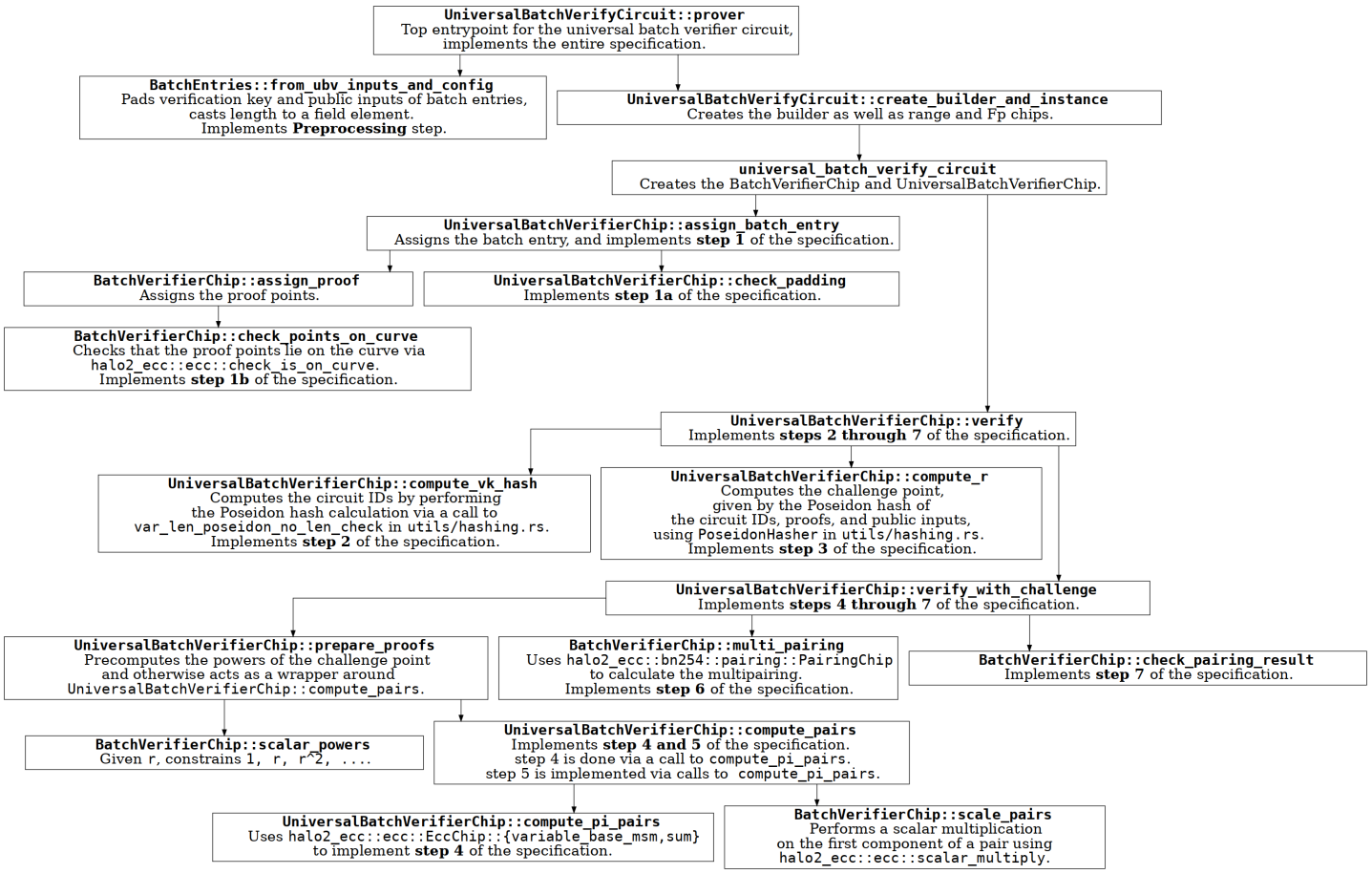


Figure 5.1: Overview of the universal batch verifier circuit functions.

Entry points

The `SafeCircuit` implementation for UniversalBatchVerifyCircuit offers the main entry points `keygen` and `prover` to generate the verification keys and to prove the universal batch verifier circuit. They handle the appropriate settings and otherwise act as a wrapper around `UniversalBatchVerifyCircuit::create_builder_and_instance`. The `prover` method also uses `BatchEntries::from_ubv_inputs_and_config` to pad inputs appropriately. This corresponds to the "Preprocessing" step in the specification.

The circuit uses the halo2-lib builder pattern, and the `GateThreadBuilder` is created in `UniversalBatchVerifyCircuit::create_builder_and_instance`, which otherwise acts as a wrapper around the `universal_batch_verify_circuit` function, creating and passing on the necessary range and finite field chips.

Step 1: Assignment and checking proof points lie on curve and padding is correct

The `UniversalBatchVerifierChip::assign_batch_entry` function is called by `universal_batch_verify_circuit` and handles **step 1** of the specification.

First, `BatchVerifierChip::assign_proof` is called, which assigns circuit variables for all the inputs and uses `BatchVerifierChip::check_points_on_curve` to verify that the proof points lie on the curve, corresponding to **step 1b** of the specification. Curve points are given in affine coordinates, and checking whether such a point lies on the curve is done using `halo2_ecc::ecc::check_is_on_curve`, which verifies the curve equation. The point at infinity is thus not representable for curve points.

Step 1a is handled by `UniversalBatchVerifierChip::assign_batch_entry` via a call to `UniversalBatchVerifierChip::check_padding`, in which the padding is constrained to be correct, using the method described in the specification.

Steps 2 and 3: Computation of circuit IDs and the challenge point

The `UniversalBatchVerifierChip::verify` function is called by `universal_batch_verify_circuit` and handles **steps 2 through 7** of the specification, split up in three calls.

First, circuit IDs are computed by calling `UniversalBatchVerifierChip::compute_vk_hash`, which performs the required Poseidon hash calculation via a call to `var_len_poseidon_no_len_check` in `utils/hashing.rs`. This implements **step 2** of the specification.

Second, the challenge point is computed by the function `UniversalBatchVerifierChip::compute_r`. The challenge point is given by another Poseidon hash, of the circuit IDs, proofs, and public inputs, and is calculated with the `PoseidonHasher` from `utils/hashing.rs`. This implements **step 3** of the specification.

Third, `UniversalBatchVerifierChip::verify_with_challenge` is called to handle **steps 4 through 7**.

Precomputation for steps 4 and 5

The `UniversalBatchVerifierChip::verify_with_challenge` function calls `UniversalBatchVerifierChip::prepare_proofs` to handle steps 4 and 5.

As step 5 involves taking the scalar multiple of curve points with a power of the challenge point, these powers are first precomputed. This is done by `BatchVerifierChip::scalar_powers`. A call to `UniversalBatchVerifierChip::compute_pairs` then handles the actual steps 4 and 5.

Step 4: Computation of S_i and the public input pairs

To perform **step 4** of the specification, `UniversalBatchVerifierChip::compute_pairs` calls `UniversalBatchVerifierChip::compute_pi_pairs`, which uses `halo2_ecc::ecc::EccChip::variable_base_msm` and `halo2_ecc::ecc::EccChip::sum` to compute the linear combinations S_i.

Step 5: Scaling of pairs for the batched pairing check

Step 5 of the specification is performed by `UniversalBatchVerifierChip::compute_pairs` directly. In order to compute the relevant scalar products, `BatchVerifierChip::scale_pairs` is used, which replaces the first component of a pair with its scalar product with a scalar using `halo2_ecc::ecc::scalar_multiply`.

Step 6: Computation of the multipairing

To compute the multipairing of all the prepared pairs in the previous steps, `UniversalBatchVerifierChip::verify_with_challenge` calls `BatchVerifierChip::multi_pairing`, which uses the `PairingChip` from `halo2_ecc::bn254::pairing` to perform this calculation. This implements **step 6** of the specification.

Step 7: Constraining the final result

For a valid batch of proofs, the result of the multipairing in the previous step should have been the neutral element of $\mathbb{G}_T$. This constraint is added by `BatchVerifierChip::check_pairing_result`, which is called by `UniversalBatchVerifierChip::verify_with_challenge`. This implements **step 7** of the specification.

5.2. Keccak circuit

The keccak circuit is specified in detail in `spec/circuits/var_len_keccak.md`.

Nebra uses a fork of Axiom's halo2-lib. Changes are made to the keccak circuit in Nebra's halo2-lib fork to complete all constraints without the need for a second phase. The proofs generated by keccak circuit are verified alongside batch verification proofs in the outer circuit. All inputs to the

Keccak Circuit and their corresponding hash images are constrained as instances. Outer verifier ensures consistency by constraining the keccak instances of bath verify and keccak proofs to be equal.

Entry points

The `SafeCircuit` implementation for KeccakCircuit offers the main entry points `keygen` and `prover` to generate the verification keys and to prove the keccak circuit. They handle the appropriate settings and otherwise act as a wrapper around `KeccakChip`.

Inputs to the circuit are formatted as `KeccakVarLenInput` as `KeccakFixedInput` which are wrapped in the `KeccakCircuitInputs` enum. `KeccakCircuit::new` serves as the main entrypoint on the prover side

Step 1: Assignment and byte decomposition of keccak inputs

`KeccakCircuit::new` is called with a keccak config and inputs as `KeccakPaddedCircuitInputs`. This function handles witness assignment and byte decomposition as specified in step 1 of the spec before passing the inputs to `KeccakChip`

Step 2, 3 and 4: Compute length in bytes, compute variable length keccak queries and compute fixed length keccak queries

`KeccakChip` provides the functions `keccak_fixed_len` and `keccak_var_len` as entrypoints. These perform step 2 3 and 4 of the spec

Step 5: Compose into field elements

Outputs computed in `KeccakChip` are composed into field elements

Precomputation for Step 6

`KeccakChip::produce_keccak_row_data` is used to compute the witness values for zkevm keccak config, which is used to constrain the actual keccak computation.

Step 6: Assign Keccak Cells

Keccak computation is assigned inside the zkevm `KeccakCircuitConfig` using the witness values generated in precomputation and constrained. Appropriate cells are exposed as instances.

Step 7 and 8: Constrained Queries

The inputs and outputs are constrained to be equal between the zkevm `KeccakCircuitConfig` and `KeccakCircuit` in `KeccakChip::constrain_fixed_queries` and `KeccakChip::constrain_var_queries`

5.3. Universal outer circuit

The universal outer circuit is specified in detail in `spec/circuits/universal_outer.md`.

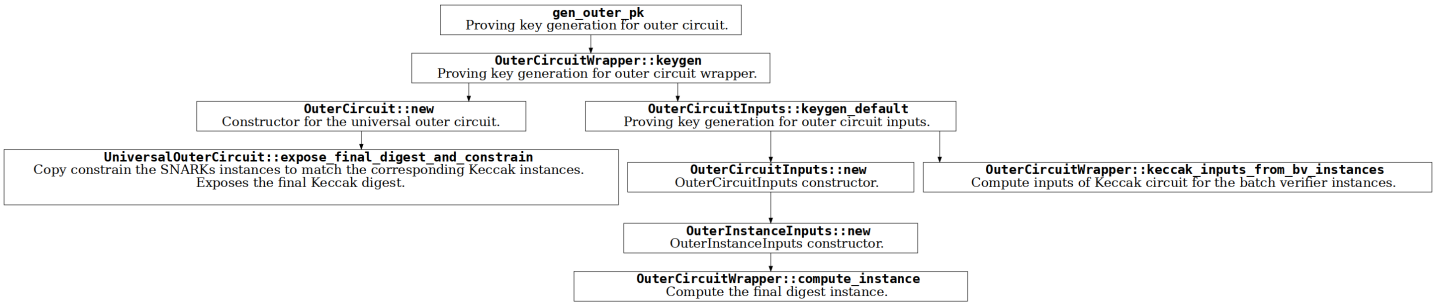


Figure 5.2: Overview of the universal outer circuit key gneration implementation.

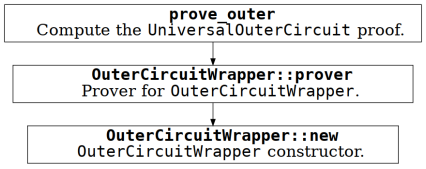


Figure 5.3: Overview of the universal outer circuit proving implementation.

Entry points

In the file `utils.rs`, the `gen_outer_pk` function is the entry point for generating the proving key of the outer circuit, whereas `prove_outer` is the entry point for the outer circuit proof generation.

Description

The circuit `UniversalOuterCircuit` implements the aggregation circuit in its `inner` field. When built by the `keygen` function, it takes as inputs the universal batch configuration and the Keccak circuit configuration thought the `OuterKeygenInputs` structure. The method `expose_final_digest_and_constrain` of `UniversalOuterCircuit` calls the Halo2 circuit builder for the `AggregationCircuit` as described in the "Snark Verifier" section of the specification. The copy constraints are added with the `constrain_equal` chip from Halo2 as described in the "Copy

Constraints" section. The final Keccak digest is exposed at the end of the function as described in the "Expose Instance" section. Finally, the verifying and proving key of the circuit are generated respectively by the Halo2 function `keygen_vk` and `keygen_pk`, and the instance size is checked.

The `prove_outer` function checks the circuit parameters and computes the outer circuit input instance, the KZG accumulator and the Keccak digest. Then, it calls the `gen_evm_proof` function from Axiom's SNARK Verifier crate to generate the outer proof.

6. Assessment Results

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

During our assessment on the scoped Universal Proof Aggregator circuits, we discovered six findings. No critical issues were found. One finding was of high impact, one was of medium impact, and the remaining findings were informational in nature.

6.1. 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.