Zellic

June 7, 2024

Prepared for
Shumo Chu
Nebra

Prepared by
Malte Leip
Sylvain Pelissier
Mohit Sharma
Zellic

Gnark support in Universal Proof Aggregation circuits

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 May 27th to June 7th, 2024. During this engagement, Zellic reviewed Gnark support in Universal Proof Aggregation circuits'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 all gnark proofs verifiable within the system parameters?
- Is completeness for vanilla Groth proofs still maintained after gnark proofs?

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

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

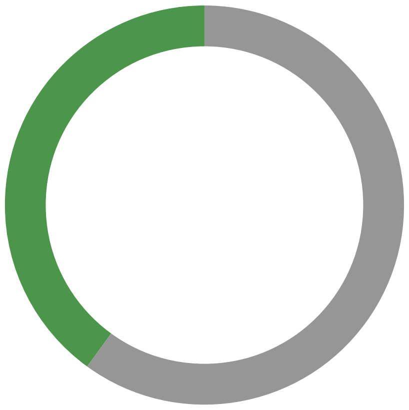
1.4. Results

During our assessment on the scoped Gnark support in Universal Proof Aggregation circuits circuits, we discovered five findings. No critical issues were found. Two findings were of low 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. ↗).

Breakdown of Finding Impacts

Impact Level	Count
🟥 Critical	0
🟧 High	0
🟨 Medium	0
🟩 Low	2
⬜ Informational	3



2. Introduction

2.1. About Gnark support in Universal Proof Aggregation circuits

Nebra contributed the following description of Gnark support in Universal Proof Aggregation circuits:

> 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.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradability weaknesses, centralization risks, and so on.

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:

Gnark support in Universal Proof Aggregation circuits Circuits

Type	Rust
Platform	Halo2

Target	Saturn
Repository	https://github.com/NebraZKP/Saturn ↗
Version	`f45412bad2bd95d70a1479eb753a5ddefc01d34c`

2.4. Project Overview

Zellic was contracted to perform a security assessment with three consultants for a total of 0.75 person-weeks. The assessment was conducted over the course of two 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.

May 28, 2024	Kick-off call
May 27, 2024	Start of primary review period
June 7, 2024	End of primary review period

3. Detailed Findings

3.1. Public input length might not be checked as intended due to overflow

Target	UniversalBatchVerifier		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The function `UniversalBatchVerifierChip::check_padding` in `circuits/src/batch_verify/universal/chip.rs` is intended to perform step 1a of the specification. This is done by constructing a bitmask that will have `1` for entries that may be arbitrary and `0` for entries that must be padding. This bitmask is obtained as follows:

```
// Bitmask computation
let max_len = entry.public_inputs.0.len();
let number_of_non_padding_elements =
    self.gate().add(ctx, entry.len, entry.has_commitment);
self.range().check_less_than_safe(
    ctx,
    number_of_non_padding_elements,
    (max_len + 1) as u64,
);
let bitmask = first_i_bits_bitmask(
    ctx,
    self.gate(),
    number_of_non_padding_elements,
    max_len as u64,
);
```

According to the specification, the intention here is to enforce that `entry.len + entry.has_commitment <= max_len`, and the left-hand side is then used for the bitmask.

However, if `entry.len = F::MODULUS - 1`, and `entry.has_commitment = 1`, then a wraparound will happen on the addition, and we will have `number_of_non_padding_elements = 0`, which will then pass the inequality check. So in that case, `UniversalBatchVerifierChip::check_padding` does not constrain `entry.len` as it should according to the specification of 1a.

Impact

We can first note that changing the `len` field of a batch entry to a different value will change the circuit ID, so it is unlikely an attacker would be able to profit from the bug described so far.

However, due to other constraints, it is actually not possible in practice to prove the UniversalBatchVerifier circuit with `entry.len = F::MODULUS - 1` and `entry.has_commitment = 1`. This is due to the function `UniversalBatchVerifierChip::constrain_commitment_hash`, which is implemented as follows:

```
fn constrain_commitment_hash(
    &self,
    ctx: &mut Context<F>,
    entry: &AssignedBatchEntry<F>,
) {
    let max_len = entry.public_inputs.0.len() as u64;
    let bitmask = ith_bit_bitmask(ctx, self.gate(), entry.len, max_len);
    let bits = bitmask.iter().map(|b| QuantumCell::<F>::from(*b));
    let lth_public_input = self.gate().inner_product(
        ctx,
        entry.public_inputs.0.iter().copied(),
        bits,
    );
    let expected =
        self.gate()
            .mul(ctx, entry.commitment_hash, entry.has_commitment);
    ctx.constrain_equal(&lth_public_input, &expected);
}
```

As `entry.len = F::MODULUS - 1`, the `bitmask` used here will be all zero, so then `lth_public_input` will also be zero. As we have `entry.has_commitment = 1`, we will then need to have `entry.commitment_hash = 0`. The rest of the circuits imply that this requires an attacker to find a keccak preimage of a multiple of `F::MODULUS`, which should be computationally infeasible.

Recommendations

We recommend ensuring the intended inequality in `UniversalBatchVerifierChip::check_padding` by ruling out the wraparound, for example by performing a range check on `entry.len` that ensures that `entry.len` must be smaller than `F::MODULUS - 1`.

Remediation

This issue has been acknowledged by Nebra, and a fix was implemented in commit 3af8a51e ↗.

3.2. Function `AssignedKeccakInputs::to_instance_values` incorrect for fixed-length inputs

Target	Keccak		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

In `circuits/src/keccak/mod.rs`, the function `AssignedKeccakInputs::to_instance_values` is implemented as follows:

```
/// Flattens `self`, disregarding the length if `self` is fixed, and keeping it
    otherwise.
pub fn to_instance_values(&self) -> Vec<AssignedValue<F>> {
    let flattening_function = match self.input_type() {
        KeccakInputType::Fixed => AssignedKeccakInput::flatten,
        KeccakInputType::Variable => {
            AssignedKeccakInput::flatten_with_len_and_commitment
        }
    };
    self.0.iter().flat_map(flattening_function).collect()
}
```

The instance should contain the commitment hash and commitment-point limbs also in the fixed case. However, `AssignedKeccakInput::flatten` is implemented as follows:

```
/// Appends the application public inputs to the vk hash.
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 behavior of `flatten` is not as one might expect from the name, as it ignores the commitment hash and commitment-point limbs.

Impact

The function `AssignedKeccakInputs::to_instance_values` will return an incorrect result in the case of fixed inputs.

The practical impact is unclear. The function is ultimately only called by `KeccakCircuit::synthesize`, and `KeccakCircuit::new` panics on fixed inputs:

```
// ...
if is_fixed {
    Self::fixed_input(ctx, &range, &mut keccak, &assigned_input);
    panic!("Keccak fixed length no longer supported");
} else {
// ...
```

Recommendations

The implementation for `AssignedKeccakInputs::to_instance_values` should be updated in the fixed case, with one possibility being disabling support for that case (i.e., panicking in that case).

The current implementation of `AssignedKeccakInput::flatten` is reasonable as used by other callers, specifically in `KeccakCircuit::var_input`, as it is used to extract the input to the proof ID keccak calculated from circuit ID and public inputs. Perhaps it makes sense to rename `flatten` for that purpose though, to reduce confusion. For example, `extract_proof_id_input` or similar might be more descriptive.

Remediation

This issue has been acknowledged by Nebra, and a fix was implemented in commit `3af8a51e` ↗.

3.3. Incorrect `len` for dummy `BatchEntry`

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

Description

The `BatchEntry::dummy` function in `circuits/src/batch_verify/universal/types.rs` file is implemented as follows:

```
/// Creates a dummy [`BatchEntry`] for `config`.
pub fn dummy(config: &UniversalBatchVerifierConfig) -> Self {
    let num_public_inputs = config.max_num_public_inputs as usize;
    let len = F::from(num_public_inputs as u64);
    // There's no difference passing `true` or `false` here.
    let has_commitment = true;
    let vk = VerificationKey::default_with_length(
        num_public_inputs,
        has_commitment,
    );
    let proof = Proof::default_with_commitment(has_commitment);
    let inputs = PublicInputs::default_with_length(num_public_inputs);
    Self {
        len,
        has_commitment,
        vk,
        proof,
        inputs,
        commitment_hash: Default::default(),
    }
}
```

In the `BatchEntry` that is returned, `len` is set to `config.max_num_public_inputs`, which is also how many entries `inputs` has. However, `has_commitments` is true, so `inputs` should also include the hash of the commitment. From the rest of the code (for example, `from_ubv_input_and_config`) it can be seen that `len` should instead be only the number of public inputs apart from the hash of the commitment.

Impact

The `BatchEntry::dummy` returns `BatchEntrys` that are malformed. Due to the way `BatchEntry::dummy` is used in practice (for key generation), this should not have any actual impact, however.

Recommendations

To ensure the return value of `BatchEntry::dummy` is consistent with the rest of the code, reduce `len` by one:

```
let num_public_inputs = config.max_num_public_inputs as usize;
 let len = F::from(num_public_inputs as u64);
 let len = F::from((num_public_inputs - 1) as u64);
```

Remediation

This issue has been acknowledged by Nebra, and a fix was implemented in commit `3af8a51e` ↗.

3.4. Incorrect assert in native `get_pairs`

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

Description

In the `get_pairs` function in `circuits/src/batch_verify/universal/native.rs`, there is the following assert:

```
assert!(
    entry.inputs.0.len()
        <= max_num_public_inputs + entry.has_commitment() as usize,
    "Too many public inputs"
);
```

Usually `max_num_public_inputs` is used for all public inputs, including the possible hash of the commitment (it is the `L` of the specification). For example, in `circuits/src/batch_verify/universal/types.rs`, the function `UniversalBatchVerifierInput::assert_consistent` has the following check:

```
assert!(
    self.inputs.0.len() + num_commitments
        <= config.max_num_public_inputs as usize,
    "Public input length exceeds maximum allowed"
);
```

Note that this check is for a `UniversalBatchVerifierInput`, which does not have the hash of the commitment appended yet. In `get_pairs`, the hash of the commitment has already been appended using `update_batch` when the check is done, so it should be as follows instead:

```
assert!(
    entry.inputs.0.len()
        <= max_num_public_inputs,
    "Too many public inputs"
);
```

Impact

The assert checks an inequality that is weaker than the inequality that should hold according to the specification.

Recommendations

We recommend to make the following change:

```
assert!(
    entry.inputs.0.len()
        <= max_num_public_inputs + entry.has_commitment() as usize,
        <= max_num_public_inputs,
    "Too many public inputs"
);
```

Remediation

This issue has been acknowledged by Nebra, and a fix was implemented in commit `3af8a51e` ↗.

3.5. Deprecated `public_input_pair` function diverged from specification

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

Description

In `circuits/src/batch_verify/universal/chip.rs`, the `UniversalBatchVerifierChip::public_input_pair` function is an older unused function implementing step 4 of the specification, having been replaced by `UniversalBatchVerifierChip::compute_pi_pairs`. The implementation of `public_input_pair` is not functionally equivalent to the current specification anymore, however.

Impact

There is no impact in the current code base, as the function is not used. The documentation comment for the function claims that it implements step 4 of the specification, which is not the case however and could lead to confusion in the future.

Recommendations

We recommend to remove this dead code or to update the documentation comments above the function, which currently claim that this function performs step 4 of the specification.

Remediation

This issue has been acknowledged by Nebra, and a fix was implemented in commit `3af8a51e` ↗.

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. Differences in batch verification to integrate gnark proofs

The subject of this audit were the protocol- and implementation-level changes to the batch verifier circuit that support verification of gnark proofs.

The differences introduced for the integration of gnark proofs are summarized below.

Types

The struct `AssignedVerificationKey` is changed to include the verifying key for the Pedersen commitment.

```rust
pub struct AssignedVerificationKey<F: EccPrimeField> {
    pub alpha: G1InputPoint<F>,
    pub beta: G2InputPoint<F>,
    pub gamma: G2InputPoint<F>,
    pub delta: G2InputPoint<F>,
    pub s: Vec<G1InputPoint<F>>,
    pub h1: G2InputPoint<F>,
    pub h2: G2InputPoint<F>,
}
```

Analogously, the struct `AssignedProof` is extended to include the Pedersen commitment and proof of knowledge.

```rust
pub struct AssignedProof<F: EccPrimeField> {
    pub a: G1InputPoint<F>,
    pub b: G2InputPoint<F>,
    pub c: G1InputPoint<F>,
    pub m: G1InputPoint<F>,
    pub pok: G1InputPoint<F>,
}
```

The separation between vanilla Groth proofs and gnark proofs is handled at the `AssignedBatchEntry` level by including a `has_commitment` boolean flag. It also contains the hash of the commitment, which is constrained to be equal to the last public input of the entry.

```
pub struct AssignedBatchEntry<F: EccPrimeField> {
    /// Assigned length of the public inputs
    pub(super) len: AssignedValue<F>,
    /// Pedersen commitment flag. Constrained to boolean values.
    pub(super) has_commitment: AssignedValue<F>,
    /// Assigned Verification Key
    pub(super) vk: AssignedVerificationKey<F>,
    /// Assigned Proof
    pub(super) proof: AssignedProof<F>,
    /// Assigned Public Inputs
    pub(super) public_inputs: AssignedPublicInputs<F>,
    /// Commitment Hash
    pub(super) commitment_hash: AssignedValue<F>,
}
```

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

In step 1a, `UniversalBatchVerifierChip::check_padding`,

1. The number of nonpadding elements in the public inputs are checked against `entry.len + entry.has_commitment` to account for the extra public input, which is constrained to the Pedersen commitment's hash.
2. Also, `UniversalBatchVerifierChip::check_padding` calls `UniversalBatchVerifierChip::check_vk_commitment_padding` and `UniversalBatchVerifierChip::check_proof_commitment_padding`, which constrain `vk.h1`, `vk.h2`, `proof.m`, and `proof.pok` to be padding if `entry.has_commitment` is false.

For step 1b, the added elements in the proof and verification key are also checked to lie on the respective curves.

For step 1c, the last public input is constrained to be equal to the hash of the commitment in the proof through a call to `UniversalBatchVerifierChip::constrain_commitment_hash`.

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

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`; `UniversalBatchVerifierChip::compute_vk_hash` now computes two hashes. One hashes the verifying key as normal, and the second adds the verifying key's commitment terms, `h1` and `h2`, as extra terms before computing the digest. The output is then selected from the two using `entry.has_commitment` as the selector.

The Fiat-Shamir step for calculating the challenge point now also hashes Pedersen terms from the proofs and verifies keys from each entry.

Steps 4: Computation of 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. The S_i pairs are computed as before, but the dot product is extended so the computed sum now includes the term `entry.has_commitment*proof.M`.

Two extra pairs for the Pedersen commitments and corresponding knowledge proofs are now also computed at this stage by making a call to `UniversalBatchVerifierChip::pedersen_pairs`.

Step 5 to 7: Computation of multipairing and constraining the final result

These steps are largely unchanged — except the addition of `m_pairs` and `pok_pairs` computed from `UniversalBatchVerifierChip::pedersen_pairs` in the previous step are also included in the pairing computation.

4.2. Terminology confusion regarding LegoSNARK

The diff reviewed for this audit introduced support for the variant of Groth16 proofs emitted by gnark (with some restrictions on the number of commitments and that commitments must be for nonpublic witnesses).

The specification describes this variant of Groth16 as LegoSNARK's "commit-and-prove" extension, as defined in the LegoSNARK paper ↗ by Matteo Campanelli, Dario Fiore, and Anaïs Querol. However, this is not accurate. Instead, gnark's variant is custom and more closely related to the paper Recursion Over Public-Coin Interactive Proof Systems; Faster Hash Verification ↗ by Alexandre Belling, Azam Soleimanian, and Olivier Bégassat. While all three schemes involve commitments to witnesses, there are also a number of differences.

4.3. Confusing implementation of `InCircuitHash` and `FieldElementRepresentation` for `AssignedVerificationKey`

While `AssignedVerificationKey` has been changed to include new fields `h1` and `h2` for verification of the proof of knowledge for the Pedersen commitment, the implementations of the `InCircuitHash`, `FieldElementRepresentation`, and `InCircuitPartialHash` have not been updated. The way these traits are used in the Poseidon implementation and the universal batch verifier chip results in behavior consistent with the specification, as handling of hashing of `h1` and `h2` is done via the extra terms that `var_len_poseidon_no_len_check_with_extra_terms` now takes.

However, these unchanged implementations of `InCircuitHash` and `FieldElementRepresentation` for `AssignedVerificationKey` are locally unexpected; for example, `FieldElementRepresentation` is documented as follows:

```
/// Interface for types that can be represented as a vector of field elements
pub trait FieldElementRepresentation<F: EccPrimeField> {
    /// Returns the representation of `self` as assigned field elements.
    fn representation(&self) -> Vec<AssignedValue<F>>;

    /// Returns the total number of field elements in `self`.
    fn num_elements(&self) -> usize;
}
```

The comments here would suggest that `representation` returns a vector of field elements that completely determines `self`, while in fact `h1` and `h2` are left out.

Given the narrow usage of `AssignedVerificationKey` in the codebase, it appears unlikely that mistakes will arise from confusion about what these two traits ensure in the near future. For best practice, we would nevertheless recommend to document that not all parts of `self` might be represented or hashed and/or possibly rename the trait.

4.4. Dummy function for `BatchEntry` has confusing comments regarding commitments

In `circuits/src/batch_verify/universal/types.rs  file`, the `BatchEntry::dummy` function is implemented and documented as follows:

```
/// Creates a dummy [`BatchEntry`] for `config`.
pub fn dummy(config: &UniversalBatchVerifierConfig) -> Self {
    let num_public_inputs = config.max_num_public_inputs as usize;
    let len = F::from(num_public_inputs as u64);
    // There's no difference passing `true` or `false` here.
    let has_commitment = true;
    let vk = VerificationKey::default_with_length(
        num_public_inputs,
        has_commitment,
    );
    let proof = Proof::default_with_commitment(has_commitment);
    let inputs = PublicInputs::default_with_length(num_public_inputs);
    Self {
        len,
        has_commitment,
        vk,
        proof,
        inputs,
        commitment_hash: Default::default(),
    }
}
```

The comment within the function says that there is no difference between passing true or false for `has_commitment`. However, both `VerificationKey::default_with_length` and `Proof::default_with_commitment` take it into account, so it is not true that there is no difference. The actual reasoning here might be as follows.

What is needed here are the padded versions. The functions that are called do not handle padding themselves however, so to ensure something padded is obtained, we pass maximum length, and also that there should be a commitment. As these default functions will use values that correspond with what is used for padding (e.g., the generator of the groups), this will return data that can be interpreted either as valid padding or as valid data for a commitment. In any case, `BatchEntry::dummy` ultimately returns a `BatchEntry` where `has_commitment` is true, so it could be useful to document that in the comment before the function as well.

4.5. Length variables could be with or without commitment

Several structs used in the codebase contain a field named `len` or similar. Before the changes reviewed by this audit, it was clear what that field should contain: the number of public inputs. However, with the support for gnark-style commitments, the hash of the commitment is an additional public input, though a public input that is computed by the code from the commitment. It is thus unclear from the name of the field alone whether `len` counts this last public input or not. This is also not clarified in the comments, for example here for `BatchEntry`:

```
/// A single entry in a batch of proofs to check.
///
/// # Note
///
/// This struct holds the *padded* verification key, proof, and public inputs,
/// as well as the original (unpadded) length as a field element. It holds a
/// boolean flag for the presence of a non-trivial Pedersen commitment. The
    circuit
/// will compute the witness from this struct. It can only be created from
/// a [`UniversalBatchVerifierInput`].
#[derive(Clone, Debug, Deserialize, Serialize)]
pub struct BatchEntry<F = Fr>
where
    F: EccPrimeField,
{
    len: F,
    has_commitment: bool,
    vk: VerificationKey,
    proof: Proof,
    inputs: PublicInputs<F>,
    commitment_hash: F,
}
```

We recommend to clarify whether `len` counts the public input from the commitment or not, with a comment or by renaming the field (e.g., to `len_without_commitment` or similar).

4.6. Outdated or incorrect comments

In some places, comments have not been updated to reflect changes to the code or include minor mistakes. We list some such instances that we came across below.

In `circuits/src/batch_verify/universal/mod.rs`, the struct `UniversalBatchVerifyCircuit` has a documenting comment that outlines what the public inputs to the circuit are, but that comment has not been updated to include the commitment hash and limbs.

Similarly, also in `circuits/src/batch_verify/universal/mod.rs`, `UniversalBatchVerifyCircuit::compute_instance` has a comment in the implementation repeating the structure of the public inputs to the circuit that is not updated.

In `circuits/src/batch_verify/universal/native.rs`, in the function `compute_pairing_check_pairs`, the comments within the function about the option

```
// Option:
// [
// ( factor * M, h1),
// ( factor * pok, h2)
// ]
```

are missing `t` (it should be `t * factor * M` and `t * factor * pok`).

In `circuits/src/keccak/utils.rs`, the function `encode_digest_as_field_elements` has comments as follows:

```
// Overflow is not possible here because:
// 2^{8*15} (highest byte decomposition power)
// * 2^8 (byte value upper bound)
// * 2 (the rest of the terms together, at most, will equal the highest one)
// = 2^130, which is strictly less than the number of bits of F, 254.
```

In the last line, the result should be `2^129` instead of `2^130`. The sentence in the last line also reads as comparing `2^130` (or now `2^129`) to `254`, while it should be the exponent that is compared. The last line should thus read as this: `= 2^129, and 129 is strictly less than the number of bits of F, 254.`

In `circuits/src/keccak/mod.rs`, the function `KeccakPaddedCircuitInput::to_instance_values` has deprecated documentation comments — similarly for `KeccakPaddedCircuitInput` and `KeccakPaddedCircuitInput::from_var_len_input`.

4.7. Confusing behavior of native `compute_challenge_points`

In `circuits/src/batch_verify/universal/native.rs`, the function `compute_challenge_points` takes a `UniversalBatchVerifierInput` as input.

The `UniversalBatchVerifierInput` type is usually used for unpadded inputs where the public inputs do not contain the hash of the commitment yet. Typically, the function `BatchEntry::from_ubv_input_and_config` in `circuits/src/batch_verify/universal/types.rs` is used to convert from `UniversalBatchVerifierInput` to `BatchEntry`, handling computation of the hash of the possible commitment and appending it to the public inputs, as well as padding.

The native function `compute_challenge_points` is intended to compute the challenge points, which are obtained as a hash that includes the public input in its padded form (with possibly the hash of the commitment). It would be consistent with other usage if `compute_challenge_points` were to use `BatchEntry::from_ubv_input_and_config` to handle the commitment hash and padding and then use the `inputs` field from the resulting `BatchEntry` for the hash.

Instead, the function's behavior differs from usage in tests and normally (via `#[cfg(test)]`), and it functions the following way:

1. During tests, it appends padding but does not append the hash of the commitment, even if a commitment is present.
2. Outside of tests, it neither appends the hash of a commitment nor adds padding.

Even though `UniversalBatchVerifierInput` is elsewhere (outside of `circuits/src/batch_verify/universal/native.rs`) used for unpadded inputs without the commitment hash, this behavior requires the caller to provide a `UniversalBatchVerifierInput` for which `inputs`, contrary to normal usage,

1. already has the commitment hash appended when in a test.
2. outside of tests, already has the commitment hash appended and is padded.

This is confusing when reading the code.

In the current codebase, it appears that `compute_challenge_points` is only called by `component`, which is a test, and `get_pairs`. The latter function is in turn only called by `component` again and by `verify_universal_groth16_batch`, and `verify_universal_groth16_batch` is only called by `test_universal_verifier_same_vk`, `test_universal_verifier_distinct_vks`, and `test_universal_verifier_general_case`, all of which are tests.

Thus, `compute_challenge_points` is only used in tests in practice, so part of the confusing duplicate behavior could be reduced by only handling the test case.

In the test case, it is still confusing why the function expects a type that usually does not contain the hash of the commitment in the public inputs to nevertheless contain it.

The reason the code produces the correct challenge points is that in both `component` and `get_pairs`, the `update_batch` function is used to add the hash of the commitment, if present, to the public inputs.

We recommend to convert the original `UniversalBatchVerifierInput` to a `BatchEntry` with `BatchEntry::from_ubv_input_and_config` instead and then use that directly in `compute_challenge_points` without adding padding. Alternatively, if there are reasons to separate appending the hash of the commitment and the padding steps, consider creating a new type for this intermediate data, and use it as input for `compute_challenge_points`. If no code changes along those lines are made, we recommend documenting the behavior and the reason why this works in comments for `compute_challenge_points` and the other relevant places.

4.8. Inaccuracies in the specification

In the variable-length keccak circuit specification `spec/circuits/var_len_keccak.md`, there is a sentence under the "Inputs" section starting with "The input consists of triples of the form", where it should now be "four-tuples" instead of "triples". The actual four-tuple is also incorrect, as it has the circuit ID first and then the length, whereas the other specs and the code have it the other way around. The same mistake also appears in the "Instance" section.

4.9. Additional check for well-constructed keccak inputs

The `KeccakPaddedCircuitInput::is_well_constructed` in `circuits/src/keccak/mod.rs` was not updated after addition of the `commitment_hash` and `commitment_point_limbs` fields to `KeccakPaddedCircuitInput`, but it would make sense to also check `self.commitment_point_limbs.len() == NUM_LIMBS * 2` here.

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 Gnark support in Universal Proof Aggregation circuits circuits, we discovered five findings. No critical issues were found. Two findings were of low impact and the remaining findings 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.