Zellic

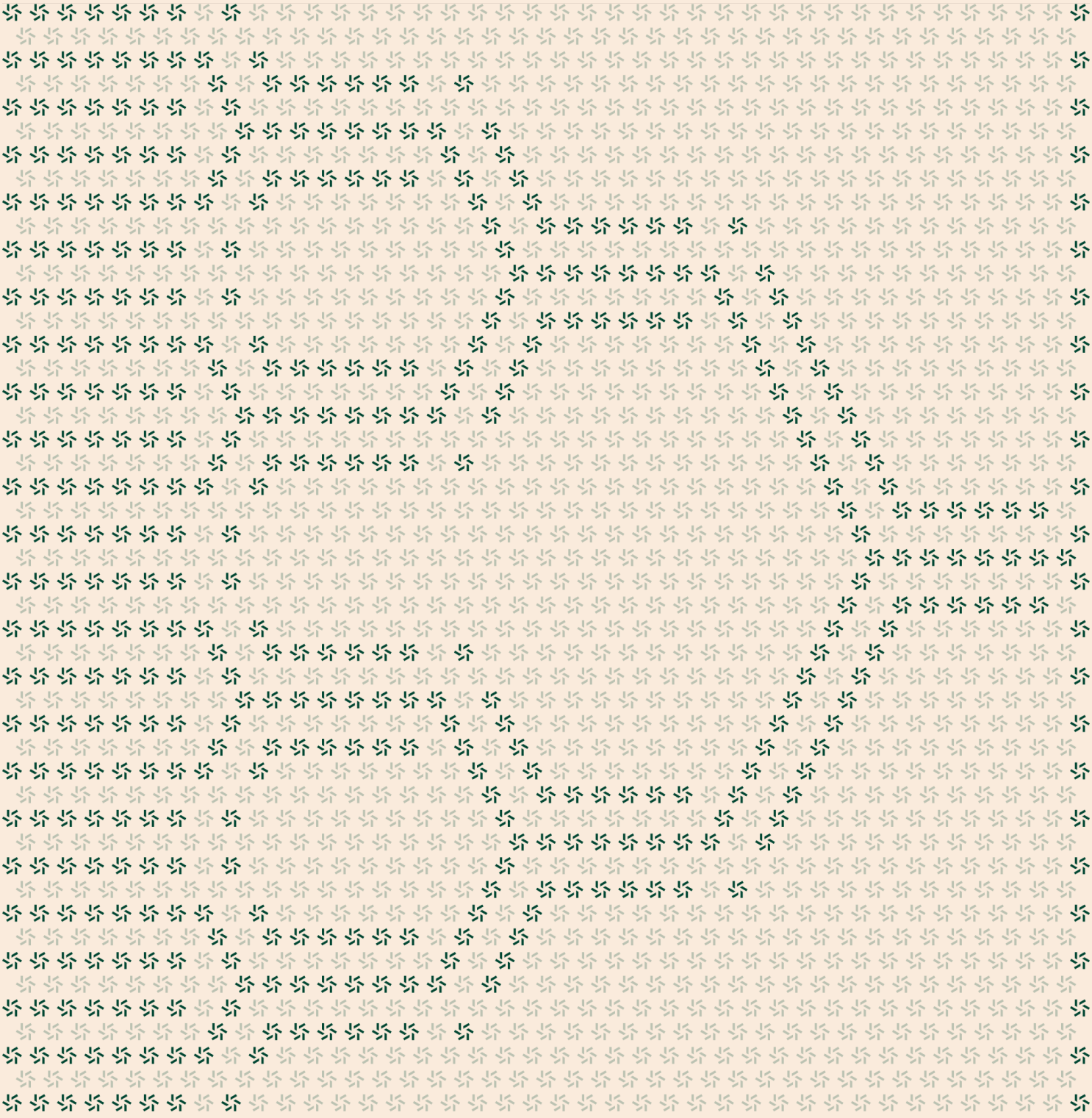
Prepared for
Mo Dong
Brevis

Prepared by
Malte Leip
Mohit Sharma
Zellic

November 13, 2024

Brevis

Zero Knowledge Security Assessment



Contents

About Zellic

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

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

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

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for Brevis from August 14th to October 23rd, 2024. During this engagement, Zellic reviewed Brevis'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:

- Does the SDK provide safe interfaces for app-circuit writers to express their business logic for computations on historical on-chain computations?
- Are there any underconstraints that make any of the circuits unsound?
- Are all circuits complete with respect to their intended specifications?

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
- On-chain components

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

Based on the number of severe findings uncovered during the audit, it is our opinion that the project is not yet ready for production. We strongly advise a comprehensive reassessment before deployment to help identify any potential issues or vulnerabilities introduced by necessary fixes or changes. We also recommend adopting a security-focused development workflow, including (but not limited to) augmenting the repository with comprehensive end-to-end tests that achieve 100% branch coverage using any common, maintainable testing framework, thoroughly documenting all function requirements, and training developers to have a security mindset while writing code.

1.4. Results

During our assessment on the scoped Brevis circuits, we discovered 35 findings. Four critical issues were found. Three were of high impact, three were of medium impact, 15 were of low impact, and the remaining findings were informational in nature.

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

Breakdown of Finding Impacts

Impact Level	Count
Critical	4
High	3
Medium	3
Low	15
Informational	10

2. Introduction

2.1. About Brevis

Brevis contributed the following description of Brevis:

> Brevis is a highly efficient ZK coprocessor that empowers smart contracts to read from the full historical on-chain data from all supported blockchains and run customizable computations in a completely trust-free way. With the power of trust-free historical data, Brevis enables exciting new use cases like data-driven DeFi, user retention and engagement features, trust-free active liquidity management, omnichain activity-based identity, and many more.

2.2. Methodology

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

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

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

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

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

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

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

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case

basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

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

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

2.3. Scope

The engagement involved a review of the following targets:

Brevis Circuits

Type	Go
Platform	gnark
Target	brevis-sdk
Repository	https://github.com/brevis-network/brevis-sdk ↗
Version	`0ed50c1a2ebd7fadaf0109ba4d5d038307fd21d5`
Programs	`sdk/*.go` `common/utils/*.go`
Target	zk-hash
Repository	https://github.com/brevis-network/zk-hash ↗
Version	`fedffbe8a45c8db46df6d67b778a8d17ae7d29d7`

Programs	`keccak/*.go`
	`mux/*.go`
	`poseidon/*.go`
	`utils/*.go`

The scope was changed while the audit was ongoing. For this reason, parts of the report may refer to repositories other than the ones listed above.

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 6.4 person-weeks. The assessment was conducted by two consultants over the course of nine calendar weeks.

Contact Information

The following project managers were associated with the engagement:

Jacob Goreski
Engagement Manager
jacob@zellic.io

Chad McDonald
Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Malte Leip
Engineer
malte@zellic.io

Mohit Sharma
Engineer
mohit@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

August 14, 2024	Start of primary review period
August 29, 2024	Meeting about scope changes
September 3, 2024	Kick-off call
October 23, 2024	End of primary review period

3. Detailed Findings

3.1. Using the `FromValues` method for the `Int248` type can lead to incorrect cached `SignBit`

Target	brevis-sdk		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

In brevis-sdk, the type `Int248` is implemented in sdk/api_int248.go, intended as an in-circuit, signed 248-bit integer type. The value is stored in the field `Val`, which is a single circuit variable holding a value that is at most 248 bits wide, with the little-endian bits of `Val` corresponding to the two's complement representation of the represented signed integer. The `Int248` type also has two additional fields, `SignBit` and `signBitSet`. The first, `SignBit`, is a circuit variable that may hold the value's sign bit, and `signBitSet` is a native Boolean that indicates whether the sign bit indicated by `SignBit` is reliable or not. The purpose of caching the sign bit this way is to reduce circuit variables and constraints needed by avoiding deconstructing `Val` into bits unnecessarily.

The method function `ensureSignBit` is used to update `SignBit` by extracting it from `Val` should `signBitSet` be false:

```
func (api *Int248API) ensureSignBit(v Int248) Int248 {
    if v.signBitSet {
        return v
    }
    bin := api.g.ToBinary(v.Val, 248)
    v.SignBit = bin[247]
    v.signBitSet = true
    return v
}
```

Whenever `signBitSet` is true, `ensureSignBit` returns immediately, thereby not requiring any additional circuit variables or constraints.

Functions operating on `Int248` that require the sign bit can then call `ensureSignBit` first and then use `SignBit` rather than computing it themselves, as can be seen, for example, in the case of `IsGreaterThan`:

```
func (api *Int248API) IsGreaterThan(a, b Int248) Uint248 {
    a = api.ensureSignBit(a)
    b = api.ensureSignBit(b)
```

```
    cmp := api.g.Cmp(a.Val, b.Val)
    isGtAsUint := api.g.IsZero(api.g.Sub(cmp, 1))

    isLt := api.g.Lookup2(
        a.SignBit, b.SignBit,
        isGtAsUint, // a, b both pos
        0, // a neg, b pos
        1, // a pos, b neg
        isGtAsUint, // a, b both neg
    )

    return newU248(isLt)
}
```

The `FromValues` method function is used to update an `Int248` by retrieving a new value from a raw circuit variable. It is implemented as follows:

```
func (v Int248) FromValues(vs ...frontend.Variable) CircuitVariable {
    if len(vs) != 1 {
        panic("Int248.FromValues only takes 1 param")
    }
    v.Val = vs[0]
    return v
}
```

This updates `Val` only, but it does not invalidate the cached sign in `SignBit`. Should `signBitSet` have been true before, then the cached sign will continue to be considered valid. However, there is no consistency check done to ensure that the cached sign bit is consistent with the sign bit of the new value.

Impact

Comparison functions such as `IsLessThan` and `IsGreaterThan` use `SignBit` to calculate the result, so an incorrect cached sign bit can lead to incorrect return values for those comparison functions.

The pattern of overwriting a previous value of an `Int248` using `FromValues` in a way that can cause this problem occurs in practice, for example, in the `Reduce` method function in sdk/datastream.go. Therefore, `Reduce` can return `Int248` with an incorrectly cached sign, leading to incorrect results when then applying `IsLessThan` or `IsGreaterThan`.

The `Reduce` function is in turn used for `MaxGeneric`, which, with a maximum, can be computed over a data stream.

The following proof of concept demonstrates how the `MaxGeneric` function using `Int248` types behaves incorrectly:

```
package sdk

import (
    "fmt"
    "math/big"
    "testing"

    "github.com/consensys/gnark-crypto/ecc"
    "github.com/consensys/gnark/frontend"
    "github.com/consensys/gnark/test"
)

type TestZellicMaxInt248FromValuesSignCircuit struct {
    g frontend.API
    i248 *Int248API
    Values [3]Int248
    ExpectedMax Int248
}

func TestZellicMaxInt248FromValuesSign(t *testing.T) {
    c := &TestZellicMaxInt248FromValuesSignCircuit{
        Values: [3]Int248{
            ConstInt248(big.NewInt(1)),
            ConstInt248(big.NewInt(47)),
            ConstInt248(big.NewInt(42)),
        },
        ExpectedMax: ConstInt248(big.NewInt(47)),
    }
    err := test.IsSolved(c, c, ecc.BN254.ScalarField())
    check(err)
}

func (c *TestZellicMaxInt248FromValuesSignCircuit) Define(g frontend.API)
    error {
    api := NewCircuitAPI(g)
    c.g = g
    c.i248 = newInt248API(g)

    listForDataStream := List[Int248](c.Values[:])
    datastream := NewDataStream(api,
        DataPoints[Int248]{
            Raw: listForDataStream,
            Toggles: []frontend.Variable{1, 1, 1},
        },
    )
```

```
    max := MaxGeneric(
        // We take maximum over the values set in the circuit struct.
        datastream,
        // The initial maximum is the smallest possible Int248, -(1<<247).
        ConstInt248(new(big.Int).Sub(big.NewInt(0),
    new(big.Int).Lsh(big.NewInt(1), 247))),
        c.i248.IsGreaterThan,
    )

    fmt.Println("Calculated max:")
    c.g.Println(max.SignBit, max.Val)
    fmt.Println("Expected max:")
    c.g.Println(c.ExpectedMax.SignBit, c.ExpectedMax.Val)
    c.i248.AssertIsEqual(max, c.ExpectedMax)

    return nil
}
```

In this test, it is attempted to calculate the maximum of the `Int248` values `1`, `47`, and `42` using `MaxGeneric`. As the initial maximum (so the maximum that would be returned if the list we take the maximum over were empty), we take the lowest possible `Int248`, which is the negative value -2^{247}. Because this value is negative, the `FromValues` bug will cause the current maximum value in the reduction performed by `MaxGeneric` to always have a cached sign bit indicating a negative value. As our list contains only positive values, each comparison will indicate that the new value is larger than the current maximum (being positive, while the current maximum is negative). Thus, `MaxGeneric` will incorrectly return the last value as the maximum. Additionally, the returned maximum will have a cached sign bit incorrectly indicating that this value is negative.

Running the above test outputs the following:

```
=== RUN TestZellicMaxInt248FromValuesSign
Calculated max:
(test.engine) zellic_max_int248_test.go:55 1 42
Expected max:
(test.engine) zellic_max_int248_test.go:57 0 47
--- FAIL: TestZellicMaxInt248FromValuesSign (0.01s)
panic: [assertIsEqual] 42 == 47
sdk.(*Int248API).AssertIsEqual
        api_int248.go:285
sdk.(*TestZellicMaxInt248FromValuesSignCircuit).Define
        zellic_max_int248_test.go:58
 [recovered]
        panic: [assertIsEqual] 42 == 47
sdk.(*Int248API).AssertIsEqual
        api_int248.go:285
sdk.(*TestZellicMaxInt248FromValuesSignCircuit).Define
```

```
        zellic_max_int248_test.go:58
```

The upshot is that various computations that users may do with `Int248` types can result in incorrect results.

Note that the above example also demonstrates how the incorrect result of `MaxGeneric` breaks the assumption that the result of `MaxGeneric` does not depend on the order the list is given in. If a specific order of the list is not otherwise enforced by the app circuit, this could thus allow an attacker to use the order of the data points to manipulate the return value.

Recommendations

The `FromValues` method function should set `signBitSet` to false, thereby invalidating the cached sign bit and ensuring that `ensureSignBit` will recompute the correct sign bit:

```
func (v Int248) FromValues(vs ...frontend.Variable) CircuitVariable {
    if len(vs) != 1 {
        panic("Int248.FromValues only takes 1 param")
    }
    v.Val = vs[0]
    v.signBitSet = false
    return v
}
```

Remediation

This issue has been acknowledged by Brevis, and a fix was implemented in commit `3191f301` ↗.

3.2. Unconstrained arithmetic operations

Target	brevis-sdk		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

The sdk provides multiple arithmetic types to application-circuit writers. This finding concerns underconstraints in the implementations of the arithmetic functions `Div` and `Sqrt` in multiple of these types, making them unsound. In the case of `Div`, the types `Uint32`, `Uint64`, `Uint248`, and `Uint521` are impacted. In the case of `Sqrt`, all of these except `Uint521` are impacted (the `Uint521` type does not have a `Sqrt` function). We will discuss these issues using the example of `Uint64`.

The `Div` function is implemented as follows in sdk/api_uint64.go:

```
// Div computes the standard unsigned integer division (like Go) and returns the
// quotient and remainder. Uses QuoRemHint
func (api *Uint64API) Div(a, b Uint64) (quotient, remainder Uint64) {
    out, err := api.g.Compiler().NewHint(QuoRemHint, 2, a.Val, b.Val)
    if err != nil {
        panic(fmt.Errorf("failed to initialize Div hint instance: %s",
    err.Error()))
    }
    q, r := out[0], out[1]
    orig := api.g.Add(api.g.Mul(q, b.Val), r)
    api.g.AssertIsEqual(orig, a.Val)
    return newU64(q), newU64(r)
}
```

Note that hints do not introduce any constraints but only assign values to circuit variables. The first lines of this function

```
out, err := api.g.Compiler().NewHint(QuoRemHint, 2, a.Val, b.Val)
if err != nil {
    panic(fmt.Errorf("failed to initialize Div hint instance: %s",
    err.Error()))
}
q, r := out[0], out[1]
```

thus introduce no constraints and only assign `q` and `r`. The `QuoRemHint` function will correctly com-

pute these assignments so that `q` and `r` will be the quotient and remainder of dividing `a.Val` by `b.Val`, respectively. However, a malicious prover is free to replace these assignments by any other assignment.

The only constraint imposed on `q` and `r` is then `a = q*b + r`. This does not suffice to enforce that `q` and `r` are the correct quotient and remainder, however. A malicious prover may for example use any assignment for `q` of their choosing and then satisfy the constraint by setting `r = a - q*b`. Analogously, as long as `b` is nonzero, a malicious prover can choose any `r` and then set `q = (a - r) / b` to satisfy the constraint.

The `Sqrt` function is implemented as follows:

```
// Sqrt returns √a. Uses SqrtHint
func (api *Uint64API) Sqrt(a Uint64) Uint64 {
    out, err := api.g.Compiler().NewHint(SqrtHint, 1, a.Val)
    if err != nil {
        panic(fmt.Errorf("failed to initialize SqrtHint instance: %s",
    err.Error()))
    }
    return newU64(out[0])
}
```

Here, the return-circuit variable is completely unconstrained.

Impact

For `Div`, a malicious prover has one degree of freedom in setting `q` and `r` and can assign arbitrary values to one of these two circuit variables. In many use cases of `Div`, only one of the two return values will be used. For example, the `Mean` function provided for data streams in sdk/datastream.go is implemented as follows:

```
// Mean calculates the arithmetic mean over the selected fields of the data
    stream. Uses Sum.
func Mean(ds *DataStream[Uint248]) Uint248 {
    sum := Sum(ds)
    quo, _ := ds.api.Uint248.Div(sum, Count(ds))
    return quo
}
```

Only the quotient is used, and so a malicious prover can make `Mean` return any value of their choosing.

For `Sqrt`, a malicious prover can return any value of their choosing.

Recommendations

For `Div`, the remainder should be range checked to be `0 <= r < b` and `q` range checked such that `q*b` cannot overflow.

For `Sqrt`, the inequalities `out[0] * out[0] <= a < (out[0] - 1)*(out[0] - 1)` should be checked. Additionally, `out[0]` will need to be range checked to prevent the multiplications overflowing.

Remediation

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

- `b2f99463` ↗
- `a2d72edb` ↗
- `7a5246a2` ↗
- `b64e19df` ↗
- `5622d4aa` ↗
- `cf45208b` ↗

3.3. Input-uniqueness check in `CircuitAPI` is ignored

Target	brevis-sdk		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

The `CircuitAPI` offers the function `AssertInputsAreUnique`, which, according to its documentation, will ensure that every item of input data is unique so that no duplicates occur:

```
// AssertInputsAreUnique Asserts that all input data (Transaction, Receipt,
// StorageSlot) are different from each other
func (api *CircuitAPI) AssertInputsAreUnique() {
    api.checkInputUniqueness = 1
}
```

However, `api.checkInputUniqueness` is never used; the relevant code in the `Define` function for `HostCircuit` in sdk/host_circuit.go is commented out:

```
func (c *HostCircuit) Define(gapi frontend.API) error {
    // ...
    //assertInputUniqueness(gapi, c.Input.InputCommitments,
    api.checkInputUniqueness)
    // ...
}
```

Accordingly, uniqueness of inputs is not actually ensured.

Impact

Users may use `AssertInputsAreUnique` and then rely in their logic on the inputs being unique. This not actually being enforced by the circuit can then make users' circuits unsound.

For example, an application circuit may compute a weighted sum of the balances of an account for various accepted tokens, in order to compute their total holdings at a particular time. Such a circuit may use `AssertInputsAreUnique` to ensure that attackers cannot inflate their supposed total holdings by using a single balance multiple times in the sum. Due to the discussed issue, this would be ineffective, and incorrect sums could be proven by the attacker.

Recommendations

We recommend to implement the uniqueness check in the `HostCircuit`. Alternatively, remove the `AssertInputsAreUnique` function from the `CircuitAPI`.

Remediation

This issue has been acknowledged by Brevis, and a fix was implemented in commit `78b7d992` ↗.

3.4. Function `assertInputUniqueness` is unsound

Target	brevis-sdk		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

The `assertInputUniqueness` function in sdk/host_circuit.go is passed a list of circuit variables `in` and the integer (assumed to be zero or one) `shouldCheck`. The intention is that if `shouldCheck` is one, then the function will constrain each nonzero component of `in` to only occur once (i.e., there should be no duplicate values within `in` except zero).

The function is implemented as follows:

```
func assertInputUniqueness(api frontend.API, in []frontend.Variable,
    shouldCheck int) {
    multicommit.WithCommitment(api, func(api frontend.API, gamma
    frontend.Variable) error {
        sorted, err := api.Compiler().NewHint(SortHint, len(in), in...)
        if err != nil {
            panic(err)
        }
        // Grand product check. Asserts the following equation holds:
        // ∑_{a \in in} a+γ = ∑_{b \in sorted} b+γ
        var lhs, rhs frontend.Variable = 0, 0
        for i := 0; i < len(sorted); i++ {
            lhs = api.Mul(lhs, api.Add(in[i], gamma))
            rhs = api.Mul(rhs, api.Add(sorted[i], gamma))
        }
        api.AssertIsEqual(lhs, rhs)

        for i := 0; i < len(sorted)-1; i++ {
            a, b := sorted[i], sorted[i+1]
            // are both a and b zero? if yes, then it's valid; if not, then
    they must be different
            bothZero := api.Select(api.IsZero(a), api.IsZero(b), 0)
            isDifferent := api.Sub(1, api.IsZero(api.Sub(a, b)))
            isValid := api.Select(bothZero, 1, isDifferent)
            isValid = api.Select(shouldCheck, isValid, 1)
            api.AssertIsEqual(isValid, 1)
        }
```

```
        return nil
    }, in...)
}
```

The overall idea behind this implementation is as follows. Another list `sorted` of same length as `in` is obtained from a hint. Hints allow the prover to provide assignments, but they do not introduce any constraints. This list `sorted` is intended to contain the same components as `in` but permuted so as to make `sorted` sorted, as the name suggests. Two things need to be checked to constrain `sorted` to actually be the sorted permutation of `in`: First, it needs to be enforced that `sorted` is a permutation of `in`, or equivalently, that value occurring in `sorted` also occur in `in`, and with the same multiplicity. Second, `sorted` must be constrained to be sorted.

Once `sorted` is known to be a sorted permutation of `in`, we can check uniqueness of nonzero entries of `sorted` instead of doing the same for `in`. As `sorted` is sorted, this is easier, as it suffices to check that two successive entries are different or both zero.

The line

```
sorted, err := api.Compiler().NewHint(SortHint, len(in), in...)
```

in the implementation fetches the sorted list from a hint, with no constraints introduced on `sorted` so far.

The logic following this

```
    // Grand product check. Asserts the following equation holds:
// ∑_{a \in in} a+γ = ∑_{b \in sorted} b+γ
var lhs, rhs frontend.Variable = 0, 0
for i := 0; i < len(sorted); i++ {
    lhs = api.Mul(lhs, api.Add(in[i], gamma))
    rhs = api.Mul(rhs, api.Add(sorted[i], gamma))
}
api.AssertIsEqual(lhs, rhs)
```

then performs the check that `sorted` is a permutation of `in`. Concretely, this checks the identity

$$\prod_{x \in \texttt{in}} (a + \gamma) = \prod_{x \in \texttt{sorted}} (x + \gamma)$$

If we consider γ a variable, then we can consider both sides to be polynomials in γ. The multiset defined by `in` is precisely given by the multiset given by the additive inverses of the roots of the polynomial on the left-hand side (we consider the roots as a multiset, so count them with multiplicity), analogously for `sorted` and the polynomial on the right-hand side. Thus, to show that the `sorted` is a permutation of `in`, or equivalently that their underlying multisets are equal, it suffices to show that the two polynomials on the left- and right-hand side are equal. If the polynomials were not equal, then evaluating the polynomials at one random point has a likelihood of at most `len(in) / r` to obtain an equality, where `r` is the order of the field over which the circuit is defined (a 254-bit prime), and

we use that `len(in)` is the degree of the polynomial. As `len(in)` is much smaller than `r`, the check succeeding at a single random point implies with high likelihood that the polynomials were already equal, and hence `sorted` is a permutation of `in`. As random challenges cannot be obtained within a circuit, gnark's commitment functionality is used to obtain a random challenge derived from `in`.

The final piece of code of the implementation is the following:

```
for i := 0; i < len(sorted)-1; i++ {
    a, b := sorted[i], sorted[i+1]
    // are both a and b zero? if yes, then it's valid; if not, then they must
    be different
    bothZero := api.Select(api.IsZero(a), api.IsZero(b), 0)
    isDifferent := api.Sub(1, api.IsZero(api.Sub(a, b)))
    isValid := api.Select(bothZero, 1, isDifferent)
    isValid = api.Select(shouldCheck, isValid, 1)
    api.AssertIsEqual(isValid, 1)
}
```

This performs constraints that, as long as `shouldCheck` is one, two successive entries of `sorted` must be different or both zero.

The discussed parts of the implementation are correct. However, what is missing is checking that `sorted` is not just any permutation but a sorted permutation of `in`.

Impact

As the check for `sorted` being sorted is missing, it is possible to make invalid lists pass all constraints. As a concrete example, consider `in = [1, 1, 0, 0, 0]`. A malicious prover could use `sorted = [1, 0, 1, 0, 0]`. This is in fact a permutation of `in`, and it also satisfies that two successive entries are either different or both zero. However, `1` occurs twice in `in`, so the nonzero entries are not unique.

Thus, `assertInputUniqueness` is unsound. As long as no single value occurs more often than `len(in) / 2` times in `in`, a malicious prover can find an assignment for `sorted` that will satisfy the constraints.

This function appears to be intended to be used when application-circuit writers configure the circuit to check that inputs are unique, though this is currently commented out; see Finding 3.3. ↗.

Recommendations

Add constraints to ensure that `sorted` is sorted.

Remediation

This issue has been acknowledged by Brevis, and a fix was implemented in commit `be28c8e2` ↗.

3.5. Missing fields in calculation of verifying key hash

Target	brevis-sdk		
Category	Coding Mistakes	**Severity**	High
Likelihood	Medium	**Impact**	High

Description

In common/utils/plonk_util.go, two functions `CalculateAppVkHashForBn254` and `CalculateAppVkHashForBn254InCircuit` are used for hashing a `replonk.VerifyingKey`, the former natively, the latter in circuit. However, they are missing some fields of the verifying key: `Size`, `SizeInv`, `Generator`, `Qcp`, and `CommitmentConstraintIndexes`. Because these are not hashed, two verifying keys that differ only in those fields will hash to the same value.

Impact

An attacker, given a verifying key for a circuit and public inputs for it, may, without being able to find witnesses that satisfy the constraint system for that circuit, be able to change the above fields of the verifying key in such a way that they can produce a proof for this modified verifying key and the same public inputs. This incorrect verifying key would then still hash to the same value as the legitimate one, and thus possibly not be distinguished by the system.

In the code that was in scope for this audit, the verifying key hashes are used in communication with the prover (local server or Brevis partner flow). If these hashes are then ultimately also used to identify the proven circuit on chain, then this issue could allow attackers to get incorrect statements accepted on chain. As on-chain components were not part of this engagement, we have not checked whether this is the case.

Recommendations

We recommend to hash all fields of the verifying key. Optimally, gnark would offer a function to do this, as this would make it less likely that future gnark extensions that add fields to the verifying key reintroduce this issue.

Remediation

This issue has been acknowledged by Brevis, and a fix was implemented in commit `05e2b9f9` ↗.

3.6. Shallow copies leading to unintended side effects

Target	brevis-sdk		
Category	Coding Mistakes	**Severity**	High
Likelihood	Low	**Impact**	High

Description

Some functions provided by the sdk take in pointer-like types as arguments (possibly occurring in a nested manner, as fields of a struct) and return these pointer-like types again (usually as part of a struct). In most such instances, a deep copy is performed, with the data that is being pointed at being copied. In some cases, however, the pointer is copied instead. This results in only a single backing data, so any modification done via one of the two identical pointers will also change the data visible through the other one.

A concrete example of this occurs in the `ZipMap2` function implemented in sdk/datastream.go, which is implemented as follows:

```
// ZipMap2 zips a data stream with a list and apply the map function over the
// zipped data. The underlying toggles of the result data stream depends on the
// toggles from the source data stream. Panics if the underlying data lengths
// mismatch
// Example: ZipMap2([1,2,3], [4,5,6], mySumFunc) -> [5,7,9]
func ZipMap2[T0, T1, R CircuitVariable](
    a *DataStream[T0], b List[T1],
    zipFunc ZipMap2Func[T0, T1, R],
) *DataStream[R] {
    if la, lb := len(a.underlying), len(b); la != lb {
        panic(fmt.Errorf("cannot zip: inconsistent underlying array lengths
    %d and %d", la, lb))
    }
    res := make([]R, len(a.underlying))
    for i := range a.underlying {
        va, vb := a.underlying[i], b[i]
        res[i] = zipFunc(va, vb)
    }
    return newDataStream(a.api, res, a.toggles)
}
```

Note that data streams consist of a slice of some type `T` and a slice of the same length of Boolean toggles, which are interpreted as indicating whether the corresponding component of the data slice is enabled. So the `ZipMap2` function zips the list coming from a data stream with another list by apply-

ing a zipping function and returns a new data stream with this data and with toggles passed through from the input data stream.

Instead of copying the slice of toggles, `ZipMap2` merely reuses the slice for the return data stream. But slices are pointer-like, so the issue described above occurs.

The following test demonstrates what can happen because of this:

```
type TestZellicZip2Circuit struct {
}

func TestZellicZip2(t *testing.T) {
    c := &TestZellicZip2Circuit{}
    err := test.IsSolved(c, c, ecc.BN254.ScalarField())
    check(err)
}

func (c *TestZellicZip2Circuit) Define(g frontend.API) error {
    api := NewCircuitAPI(g)
    u248 := api.Uint248

    // The user creates some datastream...
    input := DataPoints[Uint248]{
        Raw:  newU248s([]frontend.Variable{1, 2, 3}...),
        Toggles: []frontend.Variable{1, 0, 0},
    }
    in := NewDataStream(api, input)
    in1 := newU248s([]frontend.Variable{5, 5, 5}...)

    // ... and computes a zip involving it.
    zippedOnlyFirst := ZipMap2(in, in1, func(a Uint248, b Uint248) Uint248 {
    return u248.Add(a, b) })

    // The user assumes that zippedOnlyFirst will not be mutated by changing
    the original `toggles` list, and so reuses it.
    in.underlying[0].Val = 42
    in.underlying[1].Val = 43
    in.underlying[2].Val = 44
    in.toggles[1] = 1
    in.toggles[2] = 1
    zippedFull := ZipMap2(in, in1, func(a Uint248, b Uint248) Uint248 {
    return u248.Add(a, b) })

    // However, zippedOnlyFirst has been changed as well now.
    fmt.Println("The original ZipMap2 result: Data is as expected, but toggles
    have been unintentionally modified: now data is enabled that was not
    indended to be!")
```

```
    zippedOnlyFirst.Show()
    fmt.Println("The new ZipMap2 result: Data and toggles are updated as
    expected.")
    zippedFull.Show()

    return nil
}
```

In this test, a data stream is instantiated with certain toggles disabled. After applying `ZipMap2` to this data stream, the data stream's toggles are modified by enabling some that were previously disabled. This will also enable entries of the zipped data stream, as can be seen in the output of the above test:

```
=== RUN   TestZellicZip2
The original ZipMap2 result: Data is as expected, but toggles have been
    unintentionally modified: now data is enabled that was not indended to be!
+---+------+--------+
| # | DATA | TOGGLE |
+---+------+--------+
| 0 |  6   |   1    |
| 1 |  7   |   1    |
| 2 |  8   |   1    |
+---+------+--------+
The new ZipMap2 result: Data and toggles are updated as expected.
+---+------+--------+
| # | DATA | TOGGLE |
+---+------+--------+
| 0 |  47  |   1    |
| 1 |  48  |   1    |
| 2 |  49  |   1    |
+---+------+--------+
```

There are also other parts of the codebase where such shallow copies may be surprising to users, such as the `ZipMap3` function that is analogous to `ZipMap2`, the `Clone` function for `CircuitInput` in sdk/circuit_input.go, or functions like `NewDataStream`, though in the latter case, this behavior might be slightly less surprising.

Impact

Users that are unaware of the shallow copies might unintentionally change data that they did not wish to change.

Let us describe one particular hypothetical scenario where a user may make some choices that can result in significant problems due to the above.

In this scenario, the application developer is working with a list of some type of data. Some part

of that data (referred from now as part A) may be what is to be used for some purpose. But then there are some additional entries (part B) that should also be included in some other computations. Finally, there may be some padding to a constant length. The developer may want to make use of the `DataStream` type to organize their data. For part A, they may use such a data stream with toggles enabling only those entries that are needed for part A, and analogously they would want a second data stream with toggles enabling the entries of part A as well as those for part B. In pseudocode, part of their circuit may look like this:

```
// This function gets some data (consisting ultimately of circuit variables),
    as well as circuit variables that should be Boolean and indicate which
    entries of data are part A and which are part B.
// We assume that part A data is something that the prover is not supposed to
    be able to choose, so a commitment is exposed that will be verified by the
    verifier. But the additional part B data entries are more freely choosable
    witnesses by the prover.
func complexABComputation(data []T, togglesA []frontend.Variable, togglesB
    []frontend.Variable) (/* return types */) {
    // Checks that togglesA[i] and togglesB[i] are Boolean and not both
    enabled.
    // ...

    toggles := make([]frontend.Variable, len(togglesA))
    copy(toggles, togglesA)
    input := DataPoints[T]{
        Raw: data // circuit variables with the data
        Toggles: toggles // circuit variables reflecting which entries are
    part A
    }
    ds := NewDataStream(api, input)
    // To ensure malicious provers can not cheat with the part A data, we will
    expose a commitment that commits to the toggles for part A, and commits to
    the A-enabled data entries.
    commitment := calculateCommitment(ds.underlying, ds.toggles)
    // `commitment` will be some kind of commitment that is constrained
    properly with regards to the circuit variables data[i] and togglesA[i]

    resultA := ZipMap2(ds, someOtherData, someFunction)
    // A result was now computed based on the datastream.
    // The intention is that only those entries of the resulting datastream
    will be enabled that held that were A-entries, as encoded in the toggles
    commited to with `commitment`.
    // If nothing further was done, this would be the case.


    // Now the developer wants to work with the datastream with part B enabled
    as well. They do not need to do further computations with the original
    A-data anymore, and they do not know that `ZipMap2` reused the toggles
```

```
    slice rather than copying it. They thus believe it is fine to assign new
    circuit variables to the entries of the toggles slice.
    for i := 0; i < len(ds.toggles); i++ {
        ds.toggles[i] = api.Or(togglesA[i], togglesB[i])
    }
    // However, now resultA will have toggles that use the circuit variables
    just created (api.Or(togglesA[i], togglesB[i])) rather than the original
    circuit variables togglesA[i]!
    // This makes resultA incorrect.

    resultB := // some computation using ds

    return commitment, resultA, resultB
    // The caller will ultimately expose commitment, and use resultA and
    resultB.
}
```

In the example outlined above, a malicious prover may assign some part B entries specially crafted data that will cause computations done with `resultsA` (where these entries are unintendedly enabled) to result in a profitable outcome for them. The commitment, which is supposed to prevent this, is ineffective, as it only commits to the entries and toggles for part A. Thus, this could amount to a critical vulnerability in this hypothetical application circuit.

The above example is hypothetical; however, it does seem plausible that usage with issues such as the described one could occur.

Recommendations

We recommend to make the sdk more robust against these kind of problems by deep copying. We have not done an exhaustive search for shallow copies, so the list given in the description subsection above should not be taken to be complete. In those cases, where public functions perform shallow copies of pointer-like types and the implementation is not changed, we recommend to clearly document this to avoid users making the discussed mistake.

Remediation

This issue has been acknowledged by Brevis. In d298cb00 ↗, `ZipMap2` and `ZipMap3` were modified to copy rather than reuse the toggles slice.

3.7. Function `ConstInt248` allows invalid ranges for arguments, returning incorrect values

Target	brevis-sdk		
Category	Coding Mistakes	**Severity**	High
Likelihood	Medium	**Impact**	High

Description

The sdk provides a type `Int248` in sdk/api_int248.go for 248-bit signed integers with which the integer range from -2^{247} to $2^{247}-1$ can be represented. To represent negative values, two's complement is used.

The function `ConstInt248` can be used to instantiate an `Int248` from a native `big.Int` value, and it is implemented as follows:

```
// ConstInt248 initializes a constant Int248. This function does not generate
// circuit wires and should only be used outside of circuit. The input big int
// can be negative
func ConstInt248(v *big.Int) Int248 {
    if v.BitLen() > 248 {
        panic("cannot initialize Int248 with bit length > 248")
    }

    abs := new(big.Int).Abs(v)
    absBits := decomposeBitsExact(abs)

    if v.Sign() < 0 {
        bits := twosComplement(absBits, 248)
        a := recompose(bits, 1)
        return newI248(a, 1)
    }

    return newI248(abs, 0)
}
```

Note that the function checks the bit length of `v` and rejects values with a bit length larger than 248. This check does not account for the sign bit required in two's complement representation and is thereby insufficiently strict.

If `v` is nonnegative, then the value `v` is used directly as the returned `Int248`. Should the value of `v` be in the range $2^{247} <= v < 2^{248}$, then the bit length check passes, yet the returned `Int248` will

be incorrect, as the sign bit will be set, so that the returned value will be interpreted as a negative integer.

An analogous issue is present in the negative case. For negative `v`, the function decomposes the absolute value of `v` into bits (in little endian) and passes them to the function `twosComplement`, and then it recomposes the bits returned by that function.

This function `twosComplement` is implemented in sdk/utils.go. It takes as arguments `bits` and `n`. From its usage in `ConstInt248`, we can conclude that `bits` is intended to be interpreted as representing an unsigned integer in little-endian presentation, and the return value should be the bitwise representation of the negation of that integer, represented using two's complement with `n` bits in total. The function is implemented as follows:

```
func twosComplement(bits []uint, n int) []uint {
    padded := padBitsRight(bits, n, 0)
    flipped := flipBits(padded)
    a := recompose(flipped, 1)
    a.Add(a, big.NewInt(1))
    d := decomposeAndSlice(a, 1, 248)
    ret := make([]uint, len(d))
    for i, b := range d {
        ret[i] = uint(b.Uint64())
    }
    return ret
    }
```

The function pads `bits` to `n` components using `padBitsRight`. If `bits` already has more than `n` components, then this will panic; see the discussion in section 4.2. ↗. But if `bits` has length exactly `n`, then no panic will happen. However, as one bit is needed for the sign, in two's complement with `n` bits, actually only values of at most `n-1` bits width, and the edge case `-2^(n-1)`, can be represented. Except that one edge case, the results will be incorrect.

For example, if `bits` is of length `n` and consists of `1, 0, ..., 0, 1`, then the bits will be flipped to give `0, 1, ..., 1, 0`, and then 1 is added, giving `1, 1, ..., 1, 0`. This represents the positive value `2^(n-1) - 1`, however, not the intended negative value `-2^(n-1) - 1`.

There is a single value with `n` bits for which `twosComplement` will return the correct two's complement representation of the negation, namely `2^(n-1)`, or in bits `0, ..., 0, 1`. The bits returned by `twosComplement` will in this case be `0, ..., 0, 1` again, and this does represent the signed integer `-2^(n-1)` in `n` bit two's complement representation. On all other inputs with bit length `n`, the return value will be incorrect, however.

This in turn implies that `ConstInt248` will return incorrect values for arguments `v` in the range `-2^n < v < -2^(n-1)`.

Impact

The function `ConstInt248` will return an incorrect `Int248` value when it is called with a `*big.Int` value in one of the following ranges.

- $-2^{248}+1$ to $-2^{247}-1$
- 2^{247} to $2^{248}-1$

Recommendations

We recommend changing both `ConstInt248` and `twosComplement` to more strictly check the inputs. Concretely, `ConstInt248` should disallow every 248-bit length value as an input, except possibly for -2^{247}, and `twosComplement` should analogously disallow every length `n` input except possibly for `[0,0,...,0,1]`.

Remediation

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

- `4deade11` ↗
- `2cd40ae7` ↗
- `65bd9950` ↗

3.8. Conversion functions `fromInterface` and `Var2BigInt` return zero instead of erroring on unrecognized types

Target	brevis-sdk		
Category	Coding Mistakes	**Severity**	High
Likelihood	Low	**Impact**	Medium

Description

The functions `fromInterface` in sdk/utils.go and `Var2BigInt` in common/utils/compose.go are intended for conversion of various types to `*big.Int`. They are both implemented in a very similar way. Below is a snippet from `fromInterface`:

```
func fromInterface(input interface{}) *big.Int {
    if input == nil {
        return big.NewInt(0)
    }
    in := input.(interface{})
    var r big.Int
    switch v := in.(type) {
    case big.Int:
        r.Set(&v)
    case *big.Int:
        r.Set(v)
    case uint8:
        r.SetUint64(uint64(v))

    // [several more cases not shown]

    case []byte:
        r.SetBytes(v)
    }
    return &r
}
```

The functions distinguish various cases depending on the type of `input`, using appropriate logic to convert to `*big.Int`. However, the case distinction is lacking a default case. Thus, when called with an unsupported type, `fromInterface` and `Var2BigInt` will return a pointer to a `big.Int` with value zero.

Impact

The `fromInterface` function is internal but used in various functions exposed to users of the API, such as `ConstUint32`, `ConstUint64`, `ConstUint248`, and `ConstUint521`, that are used to construct objects of types provided by the API. Should a user call one of these functions with an unsupported type, no error will occur, but instead they will obtain an object most likely reflecting an incorrect value (zero instead of the intended value).

The `fromInterface` function is also used in other places, such as the implementation of `toBinary` for `Bytes32`, where similar problems can occur.

The `Var2BigInt` function with the same issue is public and used in the also public function `Byte32ToFrBits`.

Recommendations

We recommend to add a default case that panics with an error message to the implementation of `fromInterface` and `Var2BigInt`. This will require changes around `addOutput` in sdk/circuit_api.go as well; see Discussion point 4.8. ↗ for more details.

We also recommend to consider merging the two functions to avoid code duplication.

Remediation

This issue has been acknowledged by Brevis, and a fix was implemented in commit `cce8529a` ↗.

3.9. No range check in `ToBinary` of `Uint521`

Target	brevis-sdk		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

Many of the types offered by the sdk provide a `ToBinary` function to decompose the type to its bits. Generally, a value of the type is passed as the first argument, which is constrained to be at most `n` bits wide, where `n` is passed as a second argument. The least significant `n` bits are then returned. Note that the gnark API's `ToBinary` also enforces the relevant bit width of the original value.

In contrast to the other types, the `Uint521` version of `ToBinary` does not constrain the bit width of the value. It is implemented as follows:

```
// ToBinary decomposes the input v to a list (size n) of little-endian binary
    digits
func (api *Uint521API) ToBinary(v Uint521, n int) List[Uint248] {
    reduced := api.f.Reduce(v.Element)
    bits := api.f.ToBits(reduced)
    ret := make([]Uint248, n)
    for i := 0; i < n; i++ {
        ret[i] = newU248(bits[i])
    }
    return ret
}
```

As seen above, the full bit decomposition is obtained first, and then the least significant `n` bits are copied to the result. There is thus no constraint on the more significant bits.

Note that the formulation of the documentation comment could be taken to suggest that the `ToBinary` does indeed check that `v` is at most `n` bits wide, as otherwise the `n` returned bits would not amount to the full decomposition of `v` into binary digits. In particular, in connection with the behavior of the other `ToBinary` functions, the incorrect assumption on the side of users appears likely.

Impact

The `ToBinary` function for `Uint521` does not ensure that the decomposition of a value `v` into `n` bits amounts to a complete decomposition, so that `v` is at most `n` bits wide. Circuits of users incorrectly relying on this check may then be underconstrained.

Recommendations

We recommend to constrain `bits[i]` to be zero for `n <= i < len(bits)`.

Alternatively, clearly warn that this function `ToBinary` does not carry out this check.

Remediation

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

- `adfb500d` ↗
- `725d66f8` ↗

3.10. Keccak padding circuit incorrect for some input lengths

Target	zk-hash		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

The file keccak/pad.go contains the function `Pad101Bits` for in-circuit calculation of the `pad10*1` padding that is used for the Keccak hash. See Discussion section 4.12. ↗ for a brief description of this padding function.

The signature of the function is as follows:

```
func Pad101Bits(
    api frontend.API,
    inBits, inLenMin, inLenMax int,
    in []frontend.Variable, inLen frontend.Variable,
) []frontend.Variable
```

The input that is to be padded is passed as `in`. As this slice cannot have different lengths for different instances of the same circuit, it must always be of the same length. Thus, the circuit variable `inLen` contains the number of elements of `in` that should be considered as the input. Each component of `in` is not necessarily a single bit but encodes `inBits`-many bits. The number `inBits` needs to divide 8 for the function to work properly. Finally, `inLenMin` and `inLenMax` contain bounds for `inLen`, which is used to reduce unnecessary constraints. All lengths in the arguments are counted in terms of components of `in`, not in bits.

The return slice of circuit variables should be the `pad10*1` padded input. As this again must be always of the same length for different instances of the same circuit, the length will be the length necessary for the padded input if the input is of maximum length (`inLenMax` components of `inBits` bits each). After the padded input, the remaining components of the return slice will be padded with the dummy value zero.

The function begins by calculating the number of bits of the padded output:

```
outBitsLen := ((inLenMax*inBits+8)/1088 + 1) * 1088
```

This formula is incorrect, however, whenever `k*1088 - 8 < inLenMax*inBits <= k*1088 - 2` for some `k`. Take for example `inBits = 1` and `inLenMax = 1080`. Then the above formula yields `outBitsLen = 2 * 1088`, even though eight bits are enough for the padding, so `outBitsLen` could

be 1088.

A corrected formula would be

```
outBitsLen := ((inLenMax*inBits+2+1087)/1088) * 1088
```

Here, we add 2 to account for the padding, and `1087 = 1088 - 1` to ensure the division rounds up.

This mistake in itself only results in the return value of `Pad101Bits` having excessive length. However, as long as this extra unnecessary block would be filled with dummy zeroes after the real padding ended, this would only amount to an efficiency issue, as long as callers use the correct number of blocks from the result.

However, the analogous mistake also appears later when constructing the actual padding, so the actual padding will be incorrect for inputs in which there is only one byte left to a full round. The code calculating how many rounds to pad to is as follows:

```
rounds := (actualLen+8)/1088 + 1
```

Here, `actualLen` is the number of bits the data is long. If that is 1,080, the result will be 2, even though the data should be padded to only one round, as eight bits are enough for that. The correct formula would be

```
rounds := (actualLen + 2 + 1087) / 1088
```

analogously to how it was before.

One can check that the circuit `Pad101Bits` is indeed incorrect by modifying pad_test.go to use an input that is 1,080 bits long, which can be done by modifying the first couple of lines like this:

```
chunk := hexutil.MustDecode("0x1234")
    inLenNibs := 1200
    inLenNibs = 270
var data []byte
    for i := 0; i < 300; i++ {
    for i := 0; i < (inLenNibs / 4); i++ {
    data = append(data, chunk...)
}
var nibs []frontend.Variable
for _, b := range data {
    nibs = append(nibs, b>>4, b&15)
}
for i := len(nibs); i < inLenNibs; i++ {
    nibs = append(nibs, 0)
}
```

The test will succeed for `inLenNibs = 268` (1,072 bits) as well as `inLenNibs = 272` (1,088 bits) but fail for `inLenNibs = 270` (1,080 bits).

Impact

Whenever `k*1088 - 8 < inLenMax*inBits <= k*1088 - 2` for some natural number `k`, the padded data returned by the function will be incorrect. The Keccak hash calculated using this padding function will thus be incorrect as well for such inputs. The hash function resulting from this padding function combined with the Keccak implementation will still be as secure as a hash function as the correctly padded Keccak is, but the computed hashes will not match the Keccak specification or other implementations that do match the specification.

In Brevis's case, one place where Keccak is used is committing to outputs of app circuits. The `Pad101Bits` circuit is not used for this use case, however, because the output will always have the same number of bits for a particular circuit, so the padding can be computed out of circuit, as is done in the sdk's `commitOutput` function in sdk/host_circuit.go.

Hypothetically, should for example `commitOutput` use the `Pad101Bits` for padding, then for certain output lengths, the hash exposed as a public circuit variable by the circuit will not match the Keccak specification. Comparing this hash using a specification-conformant implementation of Keccak, for example in a smart contract on chain, would then result in the hashes not matching and, thereby, results that should be recognized as valid not being recognized as such.

However, `Pad101Bits` is currently not actually used in the in-scope code, except tests.

Recommendations

The calculation of `outBitsLen` should be corrected,

```
outBitsLen := ((inLenMax*inBits+8)/1088 + 1) * 1088
 outBitsLen := ((inLenMax*inBits+2+1087)/1088) * 1088
```

similarly for `rounds`:

```
rounds := (actualLen+8)/1088 + 1
    rounds := (actualLen + 2 + 1087) / 1088
```

Additionally, the code following the calculation of `rounds` that inserts the actual padding will also need to be changed to support the situation where the padding fits within one byte. This is because it currently inserts one byte with bits `10000000` (though in reversed order), then inserts some zeroes as necessary, and then the last `1`.

One way to do that might be to fill the remaining bits with zero and afterwards overwrite the right bit with `1`, so instead of

```
// insert the padding part
for ; j < actualLen+7; j++ {
    padded[i][j] = 0
}
padded[i][j] = 1
j++
for ; j < rounds*1088-8; j++ {
    padded[i][j] = 0
}
padded[i][j] = 1
j++

// populate the rest with dummy zeros
for ; j < outBitsLen; j++ {
    padded[i][j] = 0
}
```

do this:

```
      // populate the rest with zeros
for ; j < outBitsLen; j++ {
    padded[i][j] = 0
}

      // set the padding bits
      padded[i][actualLen + 7] = 1
      padded[i][rounds*1088 - 8] = 1
```

Remediation

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

- 5a281f31 ↗
- ac16551 ↗
- fc8fc444 ↗

3.11. Native Keccak padding and round-index functions incorrect

Target	zk-hash		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Low	**Impact**	Low

Description

This finding is about issues similar to those for the Keccak padding circuit that were discussed in Finding 3.10. ↗ but instead for the native implementation in keccak/periphery.go. For a brief overview of the `pad10*1` padding, see Discussion section 4.12. ↗.

The following function is intended to calculate the number of blocks that an input to the Keccak hash function of length `bitsLen` will be padded to:

```
func GetRoundIndex(bitsLen int) int {
    return (bitsLen + 8) / 1088
}
```

It has the same issue as the calculation of `rounds` in Finding 3.10. ↗, incorrectly returning the round number for bit lengths 1,080 to 1,086 (modulo 1,088). This function does not seem to be called anywhere, however.

The `PadBits101` function is the counterpart to `Pad101Bits` from Finding 3.10. ↗ but for native types:

```
func PadBits101(api frontend.API, data []frontend.Variable, maxRound int)
    []frontend.Variable {
    var padBits []frontend.Variable
    padBits = append(padBits, data...)
    var missedLen = 1088 - len(data)%1088
    var zeroBitLen = missedLen - 8*2

    padBits = append(padBits, api.ToBinary(1, 8)...) // 1000, 0000
    for i := 0; i < zeroBitLen; i++ {
        padBits = append(padBits, 0) //00...0000
    }
    padBits = append(padBits, api.ToBinary(128, 8)...) // 0000 0001

    padZeroLen := maxRound*1088 - len(padBits)

    for i := 0; i < padZeroLen; i++ {
```

```
        padBits = append(padBits, 0)
    }
    return padBits
}
```

It has similar problems to `Pad101Bits`. The padding is incorrect when `len(data) % 1088` is between 1,073 and 1,087. In that case, `zeroBitLen` will be negative, and padding will consist of first `1000 0000 0000 0001`, with the final bit always occurring at an incorrect location, and it is then filled with dummy zeroes.

Impact

The functions `GetRoundIndex` and `PadBits101` produce incorrect results for some inputs. Should they be used to calculate Keccak hashes, the result will be incorrect for such inputs.

Among the in-scope code in zk-hash and brevis-sdk, the `PadBits101` function is only used in the sdk's `SlotOfStructFieldInMapping` function in sdk/circuit_api.go, where usage is in such a way that the input will not be of a size that can cause the issue. The `GetRoundIndex` function is not used.

Recommendations

The function `GetRoundIndex` should be replaced by a corrected formula:

```
func GetRoundIndex(bitsLen int) int {
    return (bitsLen + 8) / 1088
    return (bitsLen + 1) / 1088
}
```

Two bits of padding must be added at least, but an exactly full block does not necessitate expanding to another block. Thus, we add 1 and round down on division.

The `PadBits101` function should be fixed by adding the correct number of zeroes, similarly to how it is described in Discussion section 4.12. ↗:

```
func PadBits101(api frontend.API, data []frontend.Variable, maxRound int)
    []frontend.Variable {
    var padBits []frontend.Variable
    padBits = append(padBits, data...)
    var missedLen = 1088 - len(data)%1088
    var zeroBitLen = missedLen - 8*2
    var zeroBitLen = (1088 + missedLen - 2) % 1088
    padBits = append(padBits, api.ToBinary(1, 8)...) // 1000, 0000
    padBits = append(padBits, 1) // Single bit 1
```

```
    for i := 0; i < zeroBitLen; i++ {
        padBits = append(padBits, 0) //00...0000
    }
    padBits = append(padBits, api.ToBinary(128, 8)...) // 0000 0001
    padBits = append(padBits, 1) // Single bit 1

    padZeroLen := maxRound*1088 - len(padBits)

    for i := 0; i < padZeroLen; i++ {
        padBits = append(padBits, 0)
    }
    return padBits
}
```

Remediation

This issue has been acknowledged by Brevis, and a fix was implemented in commit fc8fc444 ↗. The `GetRoundIndex` function was removed, with `GetKeccakRoundForPaddedBytes` to be used instead.

3.12. Padding incorrect for output-commitment computation

Target	brevis-sdk		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	N/A	**Impact**	Informational

Description

In sdk/host_circuit.go, the function `commitOutput` handles padding before calculating the Keccak hash of the output bits. See Discussion section 4.12. ↗ for a description of the padding used for Keccak. In the code, the implementation is as follows:

```
func (c *HostCircuit) commitOutput(bits []frontend.Variable) OutputCommitment {
    if len(bits)%8 != 0 {
        panic(fmt.Errorf("len bits (%d) must be multiple of 8", len(bits)))
    }

    rounds := len(bits)/1088 + 1
    paddedLen := rounds * 1088
    padded := make([]frontend.Variable, paddedLen)
    copy(padded, bits)

    // pad 101, start from one bit after the
    padded[len(bits)] = 1
    for i := len(bits) + 1; i < paddedLen-1; i++ {
        padded[i] = 0
    }
    padded[len(padded)-1] = 1

    // ...
}
```

The padding is incorrect whenever `len(bits) % 1088 == 1087`. In this case, only a single bit is left available in the last unfilled block, which is insufficient for the padding, as the shortest possible padding consist of two bits, 11. Thus, the correct padding would consist of the bit 1, followed by 1,087 bits 0, and finally another bit 1. The implementation appends only the single bit 1, however.

Let us go through the code in the concrete example of `len(bits) = 1087` to illustrate this. Then `len(bits) / 1088 = 1087 / 1088` will be zero, so we will have `rounds = 1`, and thus `paddedLen = 1088`. The lines `padded[len(bits)] = 1` and `padded[len(padded)-1] = 1` will then set the same bit to 1, and the padding will only consist of this single bit 1.

Impact

For `bits` satisfying `len(bits) % 1088 == 1087`, the computed padding will be incorrect, making the hash calculated by `commitOutput` different than the one calculated by the user's contract on chain.

However, in practice, the length of `bits` should always be divisible by 8. As long as this is satisfied, the edge case of `len(bits) % 1088 == 1087` cannot occur. The main output type of concern would be `Uint248`, which has functions `OutputUint` taking a user-provided bit length, and `OutputBool`, which could plausibly be outputted as a single bit. However, the former checks that the bit size chosen is divisible by 8:

```
func (api *CircuitAPI) OutputUint(bitSize int, v Uint248) {
    if bitSize%8 != 0 {
        panic("bitSize must be multiple of 8")
    }

    // ...
}
```

and the latter uses an eight-bit representation:

```
api.addOutput(api.g.ToBinary(v.Val, 8))
```

Should these implementations be changed or another output type be added, so that output bitstrings that are of length not divisible by 8 become possible, then the discussed padding error could occur.

Recommendations

The computation of `rounds` should be replaced by the following formula (or an equivalent one):

```
rounds := ((len(bits) + 1) / 1088) + 1
```

Remediation

This issue has been acknowledged by Brevis, and a fix was implemented in commit be28c8e2 ↗.

3.13. Assumptions made regarding log topics are not correct in full generality

Target	brevis-sdk		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Low	**Impact**	Low

Description

In Ethereum, log entries consist of

- the address that originated the log entry
- zero or more 32-byte log topics
- data of zero or more bytes of length

This is specified in section 4.4.1 of the Ethereum yellow paper ↗ (version 9fde3f4 from September 2nd, 2024).

The EVM offers five instructions to produce log entries, `LOG0`, `LOG1`, `LOG2`, `LOG3`, and `LOG4`, with the number indicating the number of topics attached to the log entry. There is no restriction on what the content of the topics might be, so contracts are free to use any 32 bytes they wish as a topic.

Solidity offers a higher level abstraction around logs, called events ↗. Events have a name and a list of named arguments that can be of different types. These arguments can be declared indexed or not, and the entire event can be declared anonymous or not. Emitting an event then emits a log entry that can be roughly described as follows:

1. If the event is not anonymous, then the first topic will be a hash of the event's signature.
2. The remaining topics are, in order, filled with the indexed fields of the event (the hash of the field, if the type is bigger than 32 bytes). Accordingly, anonymous events can have up to four indexed fields, whereas non-anonymous events can have only three.
3. The non-indexed fields are encoded together and used as the data part of a log entry.

Commonly, nonanonymous events are used, and direct usage of `LOGn` opcodes is rare, though it does occur, for example in the DAI contract ↗.

Brevis allows proving correctness of receipts, which may include some log entries. For the fields of a log entry, the following type is used, defined in sdk/circuit_input.go of the brevis-sdk repository:

```
// LogField represents a single field of an event.
type LogField struct {
    // The contract from which the event is emitted
    Contract Uint248
```

```
    // The event ID of the event to which the field belong (aka topics[0])
    EventID Uint248
    // Whether the field is a topic (aka "indexed" as in solidity events)
    IsTopic Uint248
    // The index of the field. For example, if a field is the second topic of a
    log, then Index is 1; if a field is the
    // third field in the RLP decoded data, then Index is 2.
    Index Uint248
    // The value of the field in event, aka the actual thing we care about,
    only 32-byte fixed length values are supported.
    Value Bytes32
}
```

Here, the value of one of the fields of an event is stored in `Value`, which is of type `Bytes32`. The first topic is stored as well, in a field called `EventID`, which is only of type `Uint248`, which can thus not describe uniquely the 32 bytes of the topic. In practice, however, it appears that instead of 248 bits, only the first six bytes (so 48 bits) of the first topic are actually stored in the `EventID`. The `pack` function in sdk/circuit_input.go mentions the following reasoning:

```
// pack packs the log fields into Bn254 scalars
// 4 + 3 * 59 = 181 bytes, fits into 6 fr vars
// 59 bytes for each log field:
// - 20 bytes for contract address
// - 6 bytes for topic (topics are 32-byte long, but we are only using
    the first 6 bytes distinguish them.
// 6 bytes gives a per-contract 1/2^48 chance of two different events having
    the same topic)
// - 1 bit for whether the field is a topic
// - 7 bits for field index
// - 32 bytes for value
func (r Receipt) pack(api frontend.API) []frontend.Variable {
    var bits []frontend.Variable
    bits = append(bits, api.ToBinary(r.BlockNum.Val, 8*4)...)

    for _, field := range r.Fields {
        bits = append(bits, api.ToBinary(field.Contract.Val, 8*20)...)
        bits = append(bits, api.ToBinary(field.EventID.Val, 8*6)...)
        bits = append(bits, api.ToBinary(field.IsTopic.Val, 1)...)
        bits = append(bits, api.ToBinary(field.Index.Val, 7)...)
        bits = append(bits, field.Value.toBinaryVars(api)...)
    }
    return packBitsToFr(api, bits)
}
```

This is likely done to save field elements in the packed representation of a `Receipt`.

The given argument applies to the case of nonanonymous events: as the first topic consists of a 256-bit hash of the event's signature, we should be able to treat the first 48 bits of this hash as another, shorter hash function, obtaining a chance of only $1/2^{48}$ that two given events have the same 48-bit hash.

However, this argument only applies for nonanonymous events emitted by Solidity contracts. For anonymous events, or log entries created without usage of Solidity events, the first topic may not consist of the output of a hash function or otherwise have a relevant low likelihood of collisions. For example, a hypothetical smart contract might use the first topic to store a contract version and event identifier, with the contract version making up the first six or more bytes. In that case, all such events will collide with each other with regards to the first six bytes of their first topic.

We are not aware of any deployed smart contracts that actually use logs in a way that would cause such collisions, however.

Impact

Storing only the first six bytes of the first topic of a log entry in order to identify event types without collision may not be safe for log entries generated by methods other than Solidity's nonanonymous events.

The precise impact of this depends on handling of log entries in Brevis's backend circuits, which were not part of the audit scope at the current phase.

Recommendations

We recommend to carefully check how the project handles log entries that are not nonanonymous events emitted by Solidity. As such logs are much rarer than nonanonymous Solidity events, it may be reasonable to reject them. If they are not rejected, but collision resistance for the first topic field is relied upon, then the entire 32 bytes should be stored, rather than just the first six bytes.

Remediation

This issue has been acknowledged by Brevis. Brevis informed us that they were keeping the current implementation, in acknowledgement of the tradeoff between saving field elements in the packed representation of a Receipt, and supporting contracts emitting rarer types of log entries, as standard Solidity events cover most of the cases that Brevis SDK supports.

3.14. Selector not constrained to be Boolean for `Select`

Target	brevis-sdk		
Category	Code Maturity	**Severity**	High
Likelihood	Low	**Impact**	Low

Description

Several of the sdk's types provide functions for Boolean operations `And`, `Or`, and `Not` as well as a `Select` function, which also expects a Boolean value as the first argument.

These functions' documentation does not indicate whether the caller should be responsible for ensuring that relevant arguments are Boolean (so either zero or one) or whether this will be done by the function itself.

In the cases of `And`, `Or`, and `Not`, this is enforced by the functions themselves. For example, `And` is implemented for `Uint32` as follows (in sdk/api_uint32.go):

```
// And returns 1 if a && b [&& other[0] [&& other[1]...]] is true, and 0
    otherwise
func (api *Uint32API) And(a, b Uint32, other ...Uint32) Uint32 {
    res := api.g.And(a.Val, b.Val)
    for _, v := range other {
        res = api.g.And(res, v.Val)
    }
    return newU32(res)
}
```

The gnark API's `And` function is ultimately used, which does have a documentation comment that could be read as suggesting that the caller should be responsible for checking that the arguments are either zero or one, but it in fact carries out that check itself:

```
// Or returns a & b
 // a and b must be 0 or 1
func (builder *builder) And(a, b frontend.Variable) frontend.Variable {
     builder.AssertIsBoolean(a)
     builder.AssertIsBoolean(b)
    res := builder.Mul(a, b)
    builder.MarkBoolean(res)
    return res
}
```

Thus, while gnark's documentation suggests that the caller of gnark's `And`, and thus the caller of brevis-sdk's `And`, needs to verify Booleaness of the arguments themselves, in fact, in the current version of gnark used by brevis-sdk, the gnark API is implemented so as to carry out this check.

The `Select` function is implemented as follows:

```
// Select returns a if s == 1, and b if s == 0
func (api *Uint32API) Select(s Uint32, a, b Uint32) Uint32 {
    return newU32(api.g.Select(s.Val, a.Val, b.Val))
}
```

Here, the gnark API is again used. With regards to `Select`, however, we need to distinguish between the two different implementations that gnark provides for the API, for different constraint systems. The `Select` implementation for R1CS constraint systems in frontend/cs/r1cs/api.go begins as follows:

```
// Select if i0 is true, yields i1 else yields i2
func (builder *builder) Select(i0, i1, i2 frontend.Variable) frontend.Variable
    {

    vars, _ := builder.toVariables(i0, i1, i2)
    cond := vars[0]

    // ensures that cond is boolean
    builder.AssertIsBoolean(cond)
```

Thus, in this case, gnark handles enforcing that the selector is indeed a Boolean.

In the case of SCS constraint systems, however, `Select` is implemented in frontend/cs/scs/api.go as follows:

```
// Select if b is true, yields i1 else yields i2
func (builder *builder) Select(b frontend.Variable, i1, i2 frontend.Variable)
    frontend.Variable {
    _b, bConstant := builder.constantValue(b)

    if bConstant {
        if !builder.IsBoolean(b) {
            panic(fmt.Sprintf("%s should be 0 or 1", builder.cs.String(_b)))
        }
        if _b.IsZero() {
            return i2
        }
        return i1
    }
```

```
    u := builder.Sub(i1, i2)
    l := builder.Mul(u, b)

    return builder.Add(l, i2)
}
```

Note that in the nonconstant case, `Select` just returns `i2 + b*(i1 - i2)`, with no checks done with regards to `b` being Boolean. Should the selector `b` not be constrained by the caller but be arbitrarily assignable by a malicious prover, then, assuming `i1 != i2`, the attacker may cause arbitrary results of `Select`. If the malicious prover wishes to obtain a value `v`, they can set the selector to `b = (v - i2) / (i1 - i2)`.

The Brevis sdk circuits appear to be instantiated using the SCS constraint system; see for example test/constraints_test.go. Thus, the caller of the `Select` function as implemented for `Uint32` and similar types provided by the sdk must enforce that the selector is either zero or one themselves. As the documentation does not state this, and similar functions, as well as `Select` with a R1CS constraint system, would not require this, there is a risk that users may not be aware that they need to constrain the selector themselves.

Impact

Users that are unaware of gnark's API implementation for SCS constraint systems may not realize that they need to constrain the selector for `Select` calls to be Boolean, thereby potentially allowing malicious provers to cause arbitrary return values.

Recommendations

We recommend to document which functions constrain arguments to be Boolean and for which it is required of the caller to ensure this. Additionally, it may be considered to add a constraint verifying this to the implementations of `Select`.

Remediation

This issue has been acknowledged by Brevis, and a fix was implemented in commit b02dc537 ↗.

3.15. Conversion from `Uint248` to `Uint521` fails for values wider than 96 bits

Target	brevis-sdk		
Category	Coding Mistakes	**Severity**	Low
Likelihood	High	**Impact**	Low

Description

The sdk provides multiple unsigned integer types, among them `Uint248` and `Uint521`, with the numbers (roughly) expressing the bit width storable in them. The `Uint521` type is in particular suggested to be used when needing to multiply `Uint248` values to be able to handle overflows that would happen when multiplying as `Uint248` directly. Due to this reason, it is important that there be a way to convert `Uint248` values, including large values, to `Uint521`.

The function `ToUint521` in sdk/circuit_api.go handles conversion of various types to `Uint521`. The following is the implementation of the case of `Uint248` inputs.

```
case Uint248:
    el := api.Uint521.f.NewElement([]variable{v.Val, 0, 0, 0, 0, 0})
    return newU521(el)
```

The circuit variable in which the `Uint248` value is stored is used as the first limb of the `Uint521`, with the other limbs set to zero. However, in the current implementation, each limb only stores up to 96 bits. When constructing the new element with `NewElement`, the limbs are range checked to ensure they do not exceed this bit width.

Impact

Attempting to convert a `Uint248` to `Uint521` will lead to unsatisfied constraints should the value be at least 2^{96}. This thus amounts to a completeness issue.

The following test demonstrates this issue:

```
func TestZellicToUInt521(t *testing.T) {
    circuit := &TestZellicToUInt521Circuit{A: new(big.Int).Lsh(big.NewInt(1),
    96)}
    err := test.IsSolved(circuit, circuit, ecc.BN254.ScalarField())
    check(err)
}

type TestZellicToUInt521Circuit struct {
```

```
    A frontend.Variable
    g frontend.API
    api *CircuitAPI
}

func (c *TestZellicToUInt521Circuit) Define(g frontend.API) error {
    c.g = g
    c.api = NewCircuitAPI(g)
    c.zellicTestToUInt521()

    return nil
}

func (c *TestZellicToUInt521Circuit) zellicTestToUInt521() {
    a := newU248(c.A)
    c.api.ToUint521(a)
}
```

Recommendations

Instead of using the circuit variable directly as the least significant limb, it should be decomposed into bits and then recomposed in groups of `f.BitsPerLimb() = 96` bits, with those values used as limbs.

Remediation

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

- c7bdc2e5 ↗
- f8faa2f9 ↗

3.16. Prover assignment will fail for custom inputs

Target	brevis-sdk		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Medium	**Impact**	Low

Description

In sdk/prover/assign.go, the `assignCustomInput` function does not work correctly for `input` not being nil. The function is passed the argument `input` of type `*sdkproto.CustomInput`. The struct `sdkproto.CustomInput` has a field `JsonBytes` that contains the data to be used as custom input. It should also be possible to pass `nil` as `input` when there is no custom input. For this, the function contains the following snippet,

```
// Support empty customInput
jsonBytes := ""
if input == nil {
    jsonBytes = "{}"
}
```

where `jsonBytes` is set to `"{}"` when `input` is `nil`. After this, `input` is not used anymore, and in particular `jsonBytes` is not updated to use `input.JsonBytes` if `input` was not `nil`. Because of this, the `TestAssignCustomInput` test in assign_test.go fails.

Impact

The function `assignCustomInput` will not work correctly when custom inputs are used. Ultimately, this function is used for assignments by `Prove` and `ProveAsync` in sdk/prover/server.go, and so proving circuits with custom inputs will fail.

Recommendations

To fix this, an else branch should be added to use the `jsonBytes` from `input` if `input` is not `nil`:

```
// Support empty customInput
jsonBytes := ""
if input == nil {
    jsonBytes = "{}"
    } else {
```

```
        jsonBytes = input.JsonBytes
}
```

Remediation

This issue has been acknowledged by Brevis, and a fix was implemented in commit 90248b80 ↗.

3.17. No domain separation for commitments

Target	brevis-sdk		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The circuits constructed using the sdk use receipt, storage, and transaction data but do not verify their correctness. Instead, they commit to these data points by hashing, and they ultimately expose a Merkle root hash of these commitments as a public input to the circuit. Correctness of the data points is then enforced by the Brevis backend circuits with the link being made through the Merkle root hash.

Calculation of the individual commitments from the receipt, storage, and transaction data is done in the `commitInput` function of sdk/host_circuit.go. The calculation is performed in the same way for each of the three types of data. The following snippet shows the relevant code for receipts:

```
   hasher, err := poseidon.NewBn254PoseidonCircuit(c.api)
if err != nil {
    return fmt.Errorf("error creating poseidon hasher instance: %s",
    err.Error())
}

hashOrZero := func(toggle frontend.Variable, vs []frontend.Variable)
    frontend.Variable {
    hasher.Reset()
    if len(vs) > 16 {
        panic(fmt.Sprintf("input is more than 16: %d", len(vs)))
    }
    for _, v := range vs {
        hasher.Write(v)
    }
    sum := hasher.Sum()
    return c.api.Select(toggle, sum, 0)
}

var inputCommits [NumMaxDataPoints]frontend.Variable
receipts := c.Input.Receipts
j := 0
for i, receipt := range receipts.Raw {
    packed := receipt.pack(c.api)
```

```
        inputCommits[j] = hashOrZero(receipts.Toggles[i], packed)
        j++
    }
```

As seen, the receipt is being packed into field elements by using the `pack` function. This method function is implemented for each of the three data types and consists essentially of converting the data making up the receipt into binary bits, concatenating them, and then converting this bit string to field elements. Back in `commitInput`, the commitment is constrained to be zero if the entry was disabled by the toggles, and otherwise it is constrained to be the Poseidon hash of the packed data.

There is no explicit domain separation between the different data types used in the commitment.

Impact

Lack of domain separation could hypothetically make collisions between input types possible. Suppose that it is possible to construct an input of type A and one of type B so that they pack to the same field elements. Then their commitments will be the same as well. If an attacker can achieve this with the input of type B being valid, then they will be able to submit this input of type B to the Brevis backend, and the Merkle root hash will pass verification. However, they will be able to instead use the invalid input of type A within the circuit. This would amount to a critical vulnerability.

In the present situation, this is not possible, however, because the three input data types are implicitly domain separated by their different lengths: the length of the packed data in field elements is 8 for receipts, 3 for storage, and 4 for transactions.

If these lengths matched, there may be other obstacles that could prevent an attacker from obtaining useful collisions in practice.

However, relying on the different lengths can be brittle in case of changes. For example, if the way the input data types get packed is changed or more input types are added in the future.

The easiest way to prevent collisions in the future is to domain separate the hashes by prepending something unique to the type. For example, each of the packed input data could be hashed after prepending a field element that is unique to the type, such as the byte string `BREVIS_INPUT_RECEIPT`. Alternatively, to minimize field elements used in the hash, one might also just prepend some unique bits before converting to field elements.

Recommendations

The current version of the code is not vulnerable to type confusion as discussed. For defense in depth and to avoid such a vulnerability on future changes, we nevertheless recommend to add explicit domain separation between the different input types.

Remediation

This issue has been acknowledged by Brevis. Brevis plans to add explicit domain separation between different input types in the future.

3.18. Function to export Groth16 proofs only works for proofs with exactly one commitment

Target	zk-utils (deprecated)		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Note regarding the deprecated target

Below we describe an issue in the zk-utils repository that was part of the originally contracted scope at the start of the audit. This repository was later deprecated and removed from the scope of the audit after we had found and reported the issue described below. We keep this finding here so that the report accurately reflects the findings we identified and reported during the auditing process. Due to the deprecation of the relevant code, no action is necessary in relation to this finding.

Description

In zk-utils, the function `ExportProof` in common/utils/groth16_util.go is used to unpack gnark's Groth16 proofs, returning various fixed-size arrays. For the commitments, it returns `commitment [2]*big.Int`, which is filled as follows:

```
commitment[0] = bn254Proof.Commitments[0].X.BigInt(new(big.Int))
commitment[1] = bn254Proof.Commitments[0].Y.BigInt(new(big.Int))
```

However, the type of `bn254Proof.Commitments` is `[]curve.G1Affine`, and a gnark proof can indeed have zero, one, or more commitments.

Impact

The `ExportProof` only works as intended for proofs that have exactly one commitment.

If instead there is no commitment, then the call will crash, due to an of out-of-bounds array access. Indeed, this is what happens when running the zk-bridge program circuits/fabric/headers/main/-main.go. The circuit in this case has no commitment, and running it results in output like the following:

```
chunk root ddc08833ebef5364d5cdedc989770becc949caf6cbdc72bba0e842d442136c06
prev hash 0301010101010101010101010101010101010101010101010101010101010102
end hash 931cf1c54d8ab666b200f2f88748a6a6ec03d3795a3a453895df015b2013b96d
```

```
chunk root ddc08833ebef5364d5cdedc989770becc949caf6cbdc72bba0e842d442136c06
prev hash 0301010101010101010101010101010101010101010101010101010101010102
end hash 931cf1c54d8ab666b200f2f88748a6a6ec03d3795a3a453895df015b2013b96d
[ddc08833ebef5364d5cdedc989770bec c949caf6cbdc72bba0e842d442136c06]
[3010101010101010101010101010101 1010101010101010101010101010102]
[931cf1c54d8ab666b200f2f88748a6a6 ec03d3795a3a453895df015b2013b96d]
1
4
compile
12:48:23 INF compiling circuit
12:48:23 INF parsed circuit inputs nbPublic=8 nbSecret=21764
12:48:53 INF building constraint builder nbConstraints=4502728
setup
gen witness
prove
12:55:30 DBG constraint system solver done
    nbConstraints=4502728 took=1415.957874
12:55:49 DBG prover done acceleration=none backend=groth16 curve=bn254
    nbConstraints=4502728 took=18987.907852
panic: runtime error: index out of range [0] with length 0

goroutine 1 [running]:
github.com/brevis-network/zk-utils/common/utils.ExportProof({0xdc2c40?,
    0xc000592000})
        /home/user/go/pkg/mod/github.com/brevis-network/zk-utils@v0.0.0-
    20240508102248-74fa43bfe5f5/common/utils/groth16_util.go:25 +0x292
main.main()
        /home/user/brevis-network-zk-
    bridge/circuits/fabric/headers/main/main.go:53 +0x5f8
exit status 2
```

Should the proof have more than one commitment, the `ExportProof` function will fail to return sufficient information to reconstruct the original proof, as the commitments after the first one will be missing. Should verification be attempted from the return values of `ExportProof` in that case, the proof will fail to verify.

Recommendations

We recommend to change `ExportProof` to support an arbitrary number of commitments. The return variable `commitment [2]*big.Int` would need to be changed to, for example, `commitments [][2]*big.Int`.

Remediation

The code related to the finding was deprecated before completion of the audit.

3.19. Incorrect constant used in `twosComplement`

Target	brevis-sdk		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

In sdk/utils.go, the function `twosComplement` is implemented as follows:

```
func twosComplement(bits []uint, n int) []uint {
    padded := padBitsRight(bits, n, 0)
    flipped := flipBits(padded)
    a := recompose(flipped, 1)
    a.Add(a, big.NewInt(1))
    d := decomposeAndSlice(a, 1, 248)
    ret := make([]uint, len(d))
    for i, b := range d {
        ret[i] = uint(b.Uint64())
    }
    return ret
}
```

During the call to `decomposeAndSlice`, the constant 248 is used where it should be `n`.

Impact

Whenever `n` is different than 248, `twosComplement` will return a result of incorrect length and possibly lose information when `n > 248`.

There are only two places where `twosComplement` is called within brevis-sdk. Both are in sdk/api_int248.go, and in those calls `n` is actually 248, so this bug is not hit in the current codebase.

Recommendations

We recommend to use `n` instead of `248` in the `decomposeAndSlice` call:

```
func twosComplement(bits []uint, n int) []uint {
    padded := padBitsRight(bits, n, 0)
```

```
    flipped := flipBits(padded)
    a := recompose(flipped, 1)
    a.Add(a, big.NewInt(1))
    d := decomposeAndSlice(a, 1, 248)
    d := decomposeAndSlice(a, 1, n)
    ret := make([]uint, len(d))
    for i, b := range d {
        ret[i] = uint(b.Uint64())
    }
    return ret
}
```

Remediation

This issue has been acknowledged by Brevis, and a fix was implemented in commit 4deade11 ↗.

3.20. Unexpected behavior for decompose-related functions on negative input

Target	brevis-sdk		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

In sdk/utils.go, the function `decomposeAndSlice` is used to decompose a `*big.Int` into a specified number of limbs. This function does not make any checks to ensure that the result fully represents the input value.

The `decomposeBig` function is a wrapper that performs a check to ensure that the bit length of the absolute value of the input is small enough so that it can be fully represented (panicking otherwise), and it is implemented as follows.

```
func decomposeBig(data *big.Int, bitSize, length uint) []*big.Int {
    if uint(data.BitLen()) > length*bitSize {
        panic(fmt.Errorf("decomposed integer (bit len %d) does not fit into
    output (bit len %d, length %d)",
            data.BitLen(), bitSize, length))
    }
    return decomposeAndSlice(data, bitSize, length)
}
```

However, `decomposeBig` does not ensure that the input is not negative. Since `decomposeAndSlice` does not handle negative values in a special way, it returns a representation that clashes with legitimate positive values. For example, if the input is `-1`, then every limb will be `2^bitSize - 1`.

For example, both printouts in the following example will print out `[15 15 15 15]`. This is the correct output for the second input, `(1 << 16) - 1`, whose four digits in base 16 are indeed 15. However, the first input of `-1` also produces this result.

```
r := *big.NewInt(-1)
fmt.Println(r.String(), "decomposes to", decomposeBig(&r, 4, 4))
r = *big.NewInt((1 << 16) - 1)
fmt.Println(r.String(), "decomposes to", decomposeBig(&r, 4, 4))
```

The `decomposeAndSlice` function should thus be avoided for negative values. One place in which this function is (ultimately) called, and which confusing behavior may arise for negative values, is the `Bytes32` type and its conversions to binary.

The `Bytes32` type is implemented in sdk/api_bytes32.go and offers two internal functions to be used for conversion to binary: `toBinaryVars` for the case of circuit variables and `toBinary` for the case of native types.

The two functions are implemented as follows:

```
func (v Bytes32) toBinaryVars(api frontend.API) []frontend.Variable {
    var bits []frontend.Variable
    bits = append(bits, api.ToBinary(v.Val[0], numBitsPerVar)...)
    bits = append(bits, api.ToBinary(v.Val[1], 32*8-numBitsPerVar)...)
    return bits
}

func (v Bytes32) toBinary() []uint {
    var bits []uint
    bits = append(bits, decomposeBits(fromInterface(v.Val[0]),
    uint(numBitsPerVar))...)
    bits = append(bits, decomposeBits(fromInterface(v.Val[1]),
    uint(32*8-numBitsPerVar))...)
    return bits
}
```

Suppose that `v.Val[0]` is assigned a negative number. If this is done as a witness, then this value would be reduced modulo the scalar field prime modulus `r`. Should the resulting value not be at most `numBitsPerVar` bits wide, then constraints introduced by `api.ToBinary` will not be satisfied, so the prover will emit an error.

In contrast, if the negative values are given as a native type such as `big.Int` and accordingly the second function is used, then the call to `decomposeBits` will ultimately result in a call to the `decomposeBig` function that was discussed above. If the absolute value is at most `numBitsPerVar` wide, then no error will be emitted.

The `decompose` function in common/utils/compose.go has the same behavior as `decomposeAndSlice`. Furthermore, it will not work as intended if its argument `bitSize` is bigger than 64. This latter issue also occurs for `decompose` from sdk/utils.go.

Impact

For negative values, the `decomposeAndSlice` and `decompose` functions return a decomposition that is likely unintended and which clashes with decompositions of positive values. As the case of negative values is not prevented by various wrapper functions, it can ultimately be reached from, for example, `Bytes32.toBinary`.

Thus, when negative values are assigned to the underlying fields of a `Bytes32`, the functions used for conversion to binary differ in their behavior depending on whether or not the variant for witnesses or the variant for native types was used. This confusing behavior could lead to mistakes.

Recommendations

We recommend making sure that the input is equal to or greater than zero in the functions `decomposeAndSlice` and `decompose`, by panicking on negative values.

For `decompose`, we also recommend to check that `bitSize <= 64` and panic otherwise.

Remediation

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

- `5a4114cc` ↗
- `9af3f96f` ↗
- `65bd9950` ↗

3.21. Mismatch between `Uint521` type and its documentation

Target	brevis-sdk		
Category	Code Maturity	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The sdk provides the type `Uint521`, implemented in sdk/api_uint521.go, which the documentation describes as supporting arithmetic up to 521 bits. However, this type is given by emulated field elements of the field with `2^521 - 1` elements. In particular, arithmetic operations are modulo `2^521 - 1`, and there is one 521-bit value (`2^521 - 1` itself) that cannot be represented as a `Uint521`.

For the various arithmetic operations, it is documented that an overflow can happen if the result is bigger than 2^{521}. However, an overflow happens already whenever the result is equal to 2^{521}.

Overflow behavior also differs from other `UintN` types, such as `Uint248`. These other types are stored in a single circuit variable, so as elements of a field with order `r`, with `r` being 254 bits wide. Operations are carried out in that field and so are reduced modulo `r`. In particular, it can happen that, for example, the sum of two `Uint248` is stored as a value that is wider than 248 bits, so outside of the range the type is intended to represent. In contrast, for `Uint521`, arithmetic is wrapping modulo `2^521 - 1`.

Impact

The `Uint521` cannot represent the full range implied by its name and documentation. This, as well as other discrepancies between the behavior of this type and its documentation and the behavior of similar types offered by the sdk, can result in incorrect usage.

Recommendations

We recommend to clearly document the range of `Uint521` and its behavior regarding overflows.

Remediation

This issue has been acknowledged by Brevis, and a fix was implemented in commit `ffea9394` ↗.

3.22. Raw data may be overwritten by index reuse

Target	brevis-sdk		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The `rawData` type in sdk/app.go is used to hold receipt, storage, and transaction data. There are two fields, a list `ordered` and a map `special`. The idea is that generally, data will be added to `ordered`, but in order to be able to pin certain data points to a specific index, they may be added to `special` instead. When converting a `rawData` to a list, the data in `special` will occur at their respective index, and the remaining unclaimed entries will be filled by the entries of `ordered`, in order.

The `add` function used to add data to a `rawData` is implemented as follows:

```
func (q *rawData[T]) add(data T, index ...int) {
    if len(index) > 1 {
        panic("no more than one index should be supplied")
    }
    if q.special == nil {
        q.special = make(map[int]T)
    }
    if len(index) == 1 {
        q.special[index[0]] = data
    } else {
        q.ordered = append(q.ordered, data)
    }
}
```

If an index is given, then `data` is inserted into the map at that index. However, it is not checked whether a value at that key might already exist. Thus, it can happen that a previous entry is overwritten.

Impact

Entries of `q.special` may be overwritten by `add`. The function `add` is an internal function, used by the public wrappers `AddReceipt`, `AddStorage`, and `AddTransaction`. The intention of the `special` map is to pin receipts, storage, and transaction data to a specific index. Because of this, reusing the same index would be a mistake.

Currently this mistake will not be caught. It could thus happen that a user of the sdk intends to pin two data points A and B but accidentally uses the same index. Only the last added of the two will then be available at that index (say B), but in circuit they will use this single data point for the use case of both A and B.

It appears likely that such mistakes would be detected during proper testing.

Recommendations

We recommend to check in `add` whether `q.special` already has a value at key `index[0]`. If this is the case, the function should panic with an error message warning about index reuse.

Remediation

This issue has been acknowledged by Brevis, and a fix was implemented in commit `7b2df59a` ↗.

3.23. Data might be arranged incorrectly by `rawData[T].list`

Target	brevis-sdk		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

As also discussed in Finding 3.22. ↗, the `rawData` type in sdk/app.go is used to hold receipt, storage, and transaction data. Raw data can be added by pinning to a specific index (stored in the field `special`) and added to a list `ordered`. The `list` function is then used to convert a `rawData` to an actual list (slice). The pinned entries from `special` will be added at the respective index, and the remaining entries will be filled with the entries from `ordered`. It is implemented as follows:

```
func (q *rawData[T]) list(max int) []T {
    var empty T
    var l []T
    ordered := q.ordered
    for i := 0; i < max; i++ {
        if q.special[i] != empty {
            l = append(l, q.special[i])
        } else if len(ordered) > 0 {
            l = append(l, ordered[0])
            ordered = ordered[1:]
        }
    }
    return l
}
```

Detection of whether `q.special` holds an entry at a particular index is done by comparing the value against the default value of the data type, which is what map access will return at uninitialized keys. This will correctly identify indexes at which no data point was added to `q.special`. However, data points that were added but happened to be the default value of that type will also be found, as false positives.

Because of this, the ultimate order of the data returned may be other than intended whenever a pinned entry has the default value.

For example, assume that there is a pinned entry A at index `0`, and there is additionally an ordered entry B. The expectation is that B occurs as the second entry and A as the first. But if A happens to have the default value, then B will actually occur as the first entry. As A is not detected as a real data entry, it will not explicitly be added, but padding with default values may ultimately act as if A had

been added as the second entry, as A also has the default value.

Entries with default value are unlikely to occur in practice. However, their occurrence is not impossible, as the default value of a `StorageData` data point would state that at the genesis block, address zero had value zero at storage slot zero, which is a sensible statement about historical Ethereum storage data.

Impact

In very rare edge cases, `list` might return incorrectly ordered data.

Recommendations

We recommend to check whether the map has a value at the relevant key rather than only checking whether the returned value is the default. This can be done by replacing the `q.special[i] != empty` check by something like the following:

```
if q.special[i] != empty {
// The second return value of a map access is a Boolean indicating whether a
    value was found at that key.
value, exists := q.special[i]
if exists {
```

Remediation

This issue has been acknowledged by Brevis, and a fix was implemented in commit `7b2df59a` ↗.

3.24. Invalid receipts with empty `LogField` entries may be assigned

Target	brevis-sdk		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The `buildLogFields` function in sdk/app.go converts log fields given in native types, packaged as `LogFieldData`, into the `LogField` type that is usable within the circuits. It is implemented as follows:

```
func buildLogFields(fs [NumMaxLogFields]LogFieldData) (fields
    [NumMaxLogFields]LogField) {
    empty := LogFieldData{}

    lastNonEmpty := fs[0]
    for i := 0; i < NumMaxLogFields; i++ {
        // Due to backend circuit's limitations, we must fill []LogField with
    valid data
        // up to NumMaxLogFields. If the user actually only wants less
    NumMaxLogFields
        // log fields, then we simply copy the previous log field in the list
    to fill the
        // empty spots.
        f := fs[i]
        if i > 0 && f == empty {
            f = lastNonEmpty
        } else {
            lastNonEmpty = f
        }
        fields[i] = LogField{
            // ...
        }
    }
    return
}
```

The comment explains that the backend requires valid data in each entry. The idea to deal with this is to populate empty entries with duplicates of a valid one. This is implemented by setting `lastNonEmpty` to the first entry before the loop, updating it to the current entry whenever it is nonempty, and assigning `lastNonEmpty` to the result at empty slots.

However, this will not work as intended if the very first entry is empty, and in this case, the first entry of the result will be empty as well.

Impact

The comment suggests that the backend will not be able to handle nonvalid `LogFieldData` entries and due to this, `buildLogFields` should ensure this does not happen. However, the current implementation fails to ensure this when the first `LogFieldData` entry is an empty entry.

This could plausibly happen if the user looked up a transaction by transaction hash that did not happen to emit any logs.

Recommendations

We recommend to check whether the first entry is empty in `buildLogFields` and panic if so, with an error message indicating that the user should not add receipts starting with an empty log field.

This will make it easier for the users to catch this issue early, before submitting to the Brevis backend, causing unnecessary costs. It may also be possible to catch this problem even earlier by filtering out such completely empty transactions.

Remediation

This issue has been acknowledged by Brevis, and a fix was implemented in commit `1b266347` ↗.

3.25. Parsing errors ignored in `GetHexArray`

Target	brevis-sdk		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The public `GetHexArray` function is implemented in common/utils/hex.go as follows:

```
func GetHexArray(hexStr string, maxLen int) (res []frontend.Variable) {
    for i := 0; i < maxLen; i++ {
        if i < len(hexStr) {
            intValue, _ := strconv.ParseInt(string(hexStr[i]), 16, 64)
            res = append(res, intValue)
        } else {
            res = append(res, 0)
        }
    }
    return
}
```

When the character being parsed with `strconv.ParseInt` is not a valid hexadecimal character, an error will be returned, and the value returned will be zero. However, `GetHexArray` ignores the error and uses the zero value anyway.

Impact

The `GetHexArray` function will use zero for characters that were not parsable as hexadecimal characters. Users calling `GetHexArray` likely do not intend this behavior, so this can lead to unintended results.

Recommendations

We recommend to check the returned error and panic if it is not `nil`.

Remediation

This issue has been acknowledged by Brevis, and a fix was implemented in commit `4b5029f4` ↗.

3.26. Incorrect elliptic curve

Target	brevis-sdk		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

In sdk/periphery.go, the three functions `ReadPkFrom`, `ReadVkFrom`, and `ReadProofFrom` use the elliptic curve `BLS12_377` instead of `BN254` as used elsewhere.

Impact

These functions to read proving keys, verifying keys, and proofs from disk will not be compatible with the rest of the system.

Recommendations

Change `BLS12_377` to `BN254` in those three functions.

Remediation

This issue has been acknowledged by Brevis, and a fix was implemented in commit `9c875ce7` ↗.

3.27. Proof submission calls callback even if submission fails

Target	brevis-sdk		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The function `SubmitProof`, implemented in sdk/app.go, can be used by the client to submit a proof.

After starting the submission, if an error occurs, the function returns immediately with the error:

```
res, err := q.gc.SubmitProof(&gwproto.SubmitAppCircuitProofRequest{
    QueryKey: &gwproto.QueryKey{
        QueryHash: hexutil.Encode(q.queryId),
        Nonce: q.nonce,
    },
    TargetChainId: q.dstChainId,
    Proof: hexutil.Encode(buf.Bytes()),
})
if err != nil {
    return fmt.Errorf("error calling brevis gateway SubmitProof: %s",
    err.Error())
}
if !res.GetSuccess() {
    return fmt.Errorf("error calling brevis gateway SubmitProof: cdoe %s, msg
    %s",
        res.GetErr().GetCode(), res.GetErr().GetMsg())
}
```

The caller of `SubmitProof` can pass a callback function that is to be called once the submission has been completed. Should such a callback function be set, `SubmitProof` thus waits using `waitFinalProofSubmitted` until the submission has been completed. Should `waitFinalProofSubmitted` not return any error, `opts.onSubmitted` is called. Should `waitFinalProofSubmitted` return an error, that error is printed, and a callback for that occasion, `opts.onError`, is called.

```
if opts.onSubmitted != nil {
    var cancel <-chan struct{}
    if opts.ctx != nil {
        cancel = opts.ctx.Done()
    }
```

```
    go func() {
        tx, err := q.waitFinalProofSubmitted(cancel)
        if err != nil {
            fmt.Println(err.Error())
            opts.onError(err)
        }
        opts.onSubmitted(tx)
    }()
}
```

However, note that in the error case, the function does not return early. Thus, `opts.onSubmitted` will still be called, even though `tx` likely does not contain useful data.

Impact

The callback `opts.onSubmitted` will be called even if waiting for the final submission failure. This callback may then carry out further actions that should only be done after successful submissions.

Recommendations

If it is intended that `opts.onSubmitted` is only called after a successfully completed submission, we recommend to return after the call to `opts.onError` in the error case:

```
if opts.onSubmitted != nil {
    var cancel <-chan struct{}
    if opts.ctx != nil {
        cancel = opts.ctx.Done()
    }
    go func() {
        tx, err := q.waitFinalProofSubmitted(cancel)
        if err != nil {
            fmt.Println(err.Error())
            opts.onError(err)
              return err
        }
        opts.onSubmitted(tx)
    }()
}
```

Remediation

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

- fc1fad78
- 81af2ba9

3.28. Incorrect specification for `MinGeneric` and `MaxGeneric`

Target	brevis-sdk		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

Functions to calculate the maximum and minimum over data streams is provided in sdk/datastream.go. The documentation and implementation of `MaxGeneric` is as follows:

```
// MaxGeneric finds out the maximum value from the data stream with the user
// defined sort function. Uses Reduce under the hood. Note if the data stream
    is
// empty (all data points are toggled off), this function returns 0.
func MaxGeneric[T CircuitVariable](ds *DataStream[T], initialMax T, gt
    SortFunc[T]) T {
    return Reduce(ds, initialMax, func(max, current T) (newMax T) {
        curGtMax := gt(current, max)
        return Select(ds.api, curGtMax, current, max)
    })
}
```

The documentation comment specifies this function as returning `0` on empty data streams. However, this function will in fact return the argument `initialMax` in that case.

An analogous mismatch between documentation and implementation is present for `MinGeneric`.

Impact

The two functions `MaxGeneric` and `MinGeneric` do not behave as the documentation states. However, this is unlikely to lead to incorrect use in practice, as users will have to pass an argument `initialMax` (or `initialMin`), so in the process of understanding what this argument does, users should become aware of this mismatch themselves.

Recommendations

We recommend to update the documentation so that it matches the implementation.

Remediation

This issue has been acknowledged by Brevis, and a fix was implemented in commit `df205d0e` ↗.

3.29. Ethereum header length checks for dynamic fields not checking `Extra`

Target	zk-bridge (deprecated)		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Note regarding the deprecated target

Below we describe an issue in the zk-bridge repository that was part of the originally contracted scope at the start of the audit. This repository was later deprecated and removed from the scope of the audit after we had found and reported the issue described below. We keep this finding here so that the report accurately reflects the findings we identified and reported during the auditing process. Due to the deprecation of the relevant code, no action is necessary in relation to this finding.

Description

This finding concerns the function `checkDynamicFieldLen` that is duplicated in

- circuits/fabric/headers/headerutil/utils.go in zk-bridge
- circuits/fabric/smt/utils.go in zk-bridge

This function is used to check that the dynamic length fields of a Ethereum block header have reasonable sizes. It is implemented as follows:

```
func checkDynamicFieldLen(h types.Header) bool {
    hasErr := 0
    hasErr += checkLen("Difficulty", len(h.Difficulty.Bytes()), 7)
    hasErr += checkLen("Number", len(h.Number.Bytes()), 8)
    hasErr += checkLen("GasLimit", getUintByteLen(h.GasLimit), 4)
    hasErr += checkLen("GasUsed", getUintByteLen(h.GasUsed), 4)
    hasErr += checkLen("Time", getUintByteLen(h.Time), 4)
    hasErr += checkLen("BaseFee", len(h.BaseFee.Bytes()), 7)
    return hasErr == 0
}
```

While it does check the other dynamic length fields, it is missing a check for `Extra`. This is a field of type `[]byte`, but it is specified by the Ethereum yellow paper that this field must be at most 32 bytes long. The function should thus also check that this is satisfied.

Impact

Headers with improperly long `Extra` fields may still make `checkDynamicFieldLen` return `true`, signifying proper dynamic field lengths. We are not aware of any caller for which the `Extra` field could be too long, however.

Recommendations

The `checkDynamicFieldLen` functions should also ensure that `Extra` is at most 32 bytes long:

```
func checkDynamicFieldLen(h types.Header) bool {
    hasErr := 0
    hasErr += checkLen("Difficulty", len(h.Difficulty.Bytes()), 7)
    hasErr += checkLen("Number", len(h.Number.Bytes()), 8)
    hasErr += checkLen("GasLimit", getUintByteLen(h.GasLimit), 4)
    hasErr += checkLen("GasUsed", getUintByteLen(h.GasUsed), 4)
    hasErr += checkLen("Time", getUintByteLen(h.Time), 4)
    hasErr += checkLen("BaseFee", len(h.BaseFee.Bytes()), 7)
    hasErr += checkLen("Extra", len(h.Extra), 32)
    return hasErr == 0
}
```

Remediation

The code related to the finding was deprecated before completion of the audit.

3.30. Endianess for `Bytes32`

Target	brevis-sdk		
Category	Code Maturity	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The `Bytes32` type (sdk/api_bytes32.go) is intended as an in-circuit representation of the similarly named Solidity type, as the documentation comment states:

```
// Bytes32 is an in-circuit representation of the solidity bytes32 type.
type Bytes32 struct {
    Val [2]frontend.Variable
}
```

A Solidity `bytes32` type is a sequence of 32 bytes. For storage and the stack, Ethereum operates with 32-byte words, so one `bytes32` is exactly one such word. Only memory is addressed bytewise. When storing a `bytes32` in memory, the first byte is stored at the lowest address. The following Solidity example demonstrates this:

```
pragma solidity ^0.8.0;

contract Example {
    function demonstrateBytes32InMemory() public pure returns (string memory)
    {
        // Initialize bytes32 with a constant example string
        bytes32 b32 = "Hello world!";

        // Overwrite the second byte (lowest address + 1) with a new value, say
    'X'
        // 'X' ASCII value is 88
        assembly {
            mstore(mload(0x40), b32) // Store b32 in memory
            mstore8(add(mload(0x40), 1), 88) // Store 0x58 (ASCII 'X') at the
    second byte position
            b32 := mload(mload(0x40)) // Load b32 back from memory
        }

        // Return the bytes32 as a string
        // Create a new bytes memory representation for string conversion
```

```
        bytes memory result = new bytes(32);
        assembly {
            // Copy the bytes32 value into the result bytes array
            mstore(add(result, 32), b32) // Store the bytes32 value in the new
    bytes array
        }

        return string(result); // Convert bytes to string and return
    }
}
```

When a type consisting of successive bytes or bits such as `bytes32` gets interpreted as an unsigned integer, endianess needs to be taken into account; one may interpret the first bytes/bit as the least significant (little endian) or most significant (big endian). Ethereum uses the big-endian convention on such conversions, as described in the first sentence of Appendix H of the yellow paper ↗.

Let us return to the `Bytes32` type that is part of the Brevis sdk. Internally, it stores its data in two circuit variables `Val[0]` and `Val[1]`. The reason two circuit variables must be used is that the finite field of prime order over which the circuit is defined has only order `r`, where `r` is a 254-bit prime, and so insufficiently large for 32 bytes (so 256 bits) of data. The most natural expectation would be that `Val[0]` will encode bytes 0 through `k` for some `k`, and `Val[1]` will encode bytes `k+1` through 31.

Let us consider the `toBinaryVars` function, which converts `Bytes32` to 256 circuit variables representing the bits making up the 32-byte–long bytestring.

```
// toBinaryVars defines the circuit that decomposes the Variables into little
    endian bits
func (v Bytes32) toBinaryVars(api frontend.API) []frontend.Variable {
    var bits []frontend.Variable
    bits = append(bits, api.ToBinary(v.Val[0], numBitsPerVar)...)
    bits = append(bits, api.ToBinary(v.Val[1], 32*8-numBitsPerVar)...)
    return bits
}
```

This implementation fits with the interpretation of `Val[0]` and `Val[1]` just given; the first `numBitsPerVar` bits, which should correspond to the first `numBitsPerVar / 8` bytes, are stored in `Val[0]`, with the remaining bytes stored in `Val[1]`. The `api.ToBinary` function decomposes field elements into little-endian bits. Thus, we come to conclusion that `Bytes32` is stored by dividing the bytes up into the first `numBitsPerVar / 8` bytes and the remainder, with the former being stored in `Val[0]` by interpreting the bytes as little-endian representation of an unsigned integer to base 256, and similarly, the latter is being stored in `Val[1]`.

The `FromBinary` fits with this interpretation:

```
func (api *Bytes32API) FromBinary(vs ...Uint248) Bytes32 {
    var list List[Uint248] = vs
```

```
    values := list.Values()
    for i := len(vs); i < 256; i++ {
        values = append(values, 0)
    }
    res := Bytes32{}
    res.Val[0] = api.g.FromBinary(values[:numBitsPerVar]...)
    res.Val[1] = api.g.FromBinary(values[numBitsPerVar:]...)
    return res
}
```

However, `ConstBytes32` functions differently:

```
// ConstBytes32 initializes a constant Bytes32 circuit variable. Panics if the
// length of the supplied data bytes is larger than 32.
func ConstBytes32(data []byte) Bytes32 {
    if len(data) > 32 {
        panic(fmt.Errorf("ConstBytes32 called with data of length %d",
    len(data)))
    }

    bits := decomposeBits(new(big.Int).SetBytes(data), 256)

    lo := recompose(bits[:numBitsPerVar], 1)
    hi := recompose(bits[numBitsPerVar:], 1)

    return Bytes32{[2]frontend.Variable{lo, hi}}
}
```

This function is passed a slice of bytes `data`. Based on what was discussed before regarding the other functions, we would expect that `data[0]` through `data[(numBitsPerVar / 8) - 1]` are stored in `Val[0]` and the remaining bytes in `Val[1]`.

Instead, the function first uses `new(big.Int).SetBytes(data)` to obtain a `big.Int` from `data`. This will interpret `data` in big endian. So `data[0]` will be the most significant byte of the resulting integer. This integer is then converted to bits with `decomposeBits`, which will order bits with little endian. Thus, now `data[0]`, as the most significant byte, will occur as bits 248 to 255. Finally, the bits are recomposed (using little-endian interpretation again) into two values. The value `lo`, used for `Val[0]`, will consist of the first `numBitsPerVar` bits, which will thus correspond to bytes `byte[31-0]` to `data[31-((numBitsPerVar / 8) - 1)]`. So the least significant eight bits of `Val[0]` will be `data[31]`, the next least significant eight bits will be `data[30]`, and so on, up to the most significant eight bits `data[31-((numBitsPerVar / 8) - 1)]`. The second field `Val[1]` will have as the least significant eight bits the byte `data[31-(numBitsPerVar / 8)]`.

This way that `ConstBytes32` handles its argument does thus not fit a compatible interpretation of the `Bytes32` data type that also incorporates the other functions; the order of the bytes is reversed by `ConstBytes32`.

It would be instructive to also look at the following function, `SlotOfArrayElement`, from sdk/circuit_api.go:

```
// SlotOfArrayElement computes the storage slot for an element in a solidity
// array state variable. arrSlot is the plain slot of the array variable.
// index determines the array index. offset determines the
// offset (in terms of bytes32) within each array element.
func (api *CircuitAPI) SlotOfArrayElement(arrSlot Bytes32, elementSize int,
    index, offset Uint248) Bytes32 {
    //api.Uint248.AssertIsLessOrEqual(offset, ConstUint248(elementSize))
    o := api.g.Mul(index.Val, elementSize)
    return Bytes32{Val: [2]variable{
        api.g.Add(arrSlot.Val[0], o, offset.Val),
        arrSlot.Val[1],
    }}
}
```

Here, slots in storage are addressed with `Bytes32`. By the Ethereum specification, these should be interpreted as big endian to convert to unsigned integers' indexing slots, in order to add the offset. However, the function adds the (assumed small) offset to `Val[0]`, which suggests that the slot address is actually stored in little endian in the `Bytes32` given as argument and will be similarly for the return value.

This is compatible with `ConstBytes32`, if a `[]byte` input is obtained by converting the address to 32 bytes using Ethereum's big-endian standard and then converted to `Bytes32` using `ConstBytes32`, which thus flips the order of the bytes. The return value of `SlotOfArrayElement` could then be compared against similar addresses also obtained in flipped representation using `ConstBytes32`. Both `ConstBytes32` and `SlotOfArrayElement` are implemented with a surprising reversion of the order of the bytes, but these cancel each other out so that they are compatible with each other.

Impact

The interface for `Bytes32` and its use is confusing and inconsistent regarding how the type is to be interpreted and in which orders the bytes are stored. This can cause mistakes when users of the sdk use this type.

The root cause of this is that the `Bytes32` is in `SlotOfArrayElement` and `ConstBytes32` used as if it were a byte of 256-bit unsigned integers. Endianess questions arise whenever one converts between a type consisting of a list of values (such as a list of bytes) and a type for numeric values. Using the same type with both interpretations makes the need for such conversions particularly confusing. The `Bytes32` type should thus not be used like this; for 256-bit unsigned integers, a `Uint256` type should be used. This would allow for explicit and therby cleaner and more transparent conversions.

Recommendations

We recommend to clearly document which functions flip orders of bytes and, on conversion, which endianess is used.

We also recommend to resolve the current discrepancy with regards to ordering of the bytes/bits between `ConstBytes32` and the `toBinaryVars` and `FromBinary` functions.

The option we would suggest would be to use a new type `Uint256` for use cases such as `SlotOfArrayElement`. It could be documented that this type stores its data in little endian. If the current `ConstBytes32` copied for use for `Uint256` were then named to something like `ConstFromBigEndianBytes`, then it would be transparent how this type behaves. As it takes arguments in big-endian order but stores data in little-endian order, it will reverse the order, which is as expected then. The `ConstBytes32` function for `Bytes32` should in this case be changed to not flip the order of the bytes, to establish compatibility with `toBinaryVars` and `FromBinary`.

An alternative would be to change `ConstBytes32` to store the first byte in `Val[0]` and the last 31 bytes in `Val[1]`. Those 31 bytes should be stored so that the least significant eight bits of `Val[1]` correspond to byte 31. If one does it this way, then `Val[0]` and `Val[1]` would be ordered in the expected way, with `Val[0]` holding lower indexed bytes than `Val[1]`, and compatibly with `toBinaryVars` and `FromBinary`, if they are changed to take into account that `Val[1]` now stores a list of bytes in big endian instead of little-endian order as before. Additionally, it would still be possible to do the addition needed in `SlotOfArrayElement` by just adding one slot (now `Val[1]`).

Remediation

This issue has been acknowledged by Brevis. In `9d80b2bf` ↗, Brevis renamed he `ConstBytes32` function to `ConstFromBigEndianBytes`.

3.31. Sign bit computed but not enabled in `Select` for `Int248`

Target	brevis-sdk		
Category	Optimization	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The `Int248` type that is part of the sdk caches the sign for efficiency reasons; see the more detailed discussion in Finding 3.1. In the `Select` function in sdk/api_int248.go, the sign bit of the result is computed based on the cached sign bits of the inputs, if they are available:

```
func (api *Int248API) Select(s Uint248, a, b Int248) Int248 {
    v := Int248{}
    v.Val = api.g.Select(s.Val, a.Val, b.Val)
    if a.signBitSet && b.signBitSet {
        v.SignBit = api.g.Select(s.Val, a.SignBit, b.SignBit)
    }
    return v
}
```

However, the cached sign bit `v.SignBit` of the result will never be used, because `v.signBitSet` will still be `false`.

Impact

When the two input values have a valid cached sign bit, constraints will be used to compute the sign bit of the result, yet in a later call to `ensureSignBit`, this cached sign bit will not be used and instead be recomputed from scratch using an expensive binary decomposition.

Recommendations

We recommend to set `v.signBitSet` to `true` in the `if` body:

```
if a.signBitSet && b.signBitSet {
    v.SignBit = api.g.Select(s.Val, a.SignBit, b.SignBit)
        v.signBitSet = true
}
```

Remediation

This issue has been acknowledged by Brevis, and a fix was implemented in commit 3191f301 ↗.

3.32. Reduction ineffective in `Values` function for `Uint521`

Target	brevis-sdk		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

In sdk/api_uint521.go of the sdk repository, the `Values` function for the `Uint521` type is implemented as follows:

```
func (v Uint521) Values() []frontend.Variable {
    u521Field.Reduce(v.Element)
    return v.Limbs
}
```

The call to `Reduce` will not change `v.Element` but only return the reduced value.

Impact

The current implementation does not match the intention to return the reduced limbs.

Recommendations

To return the reduced limbs, the function should be implemented as follows:

```
func (v Uint521) Values() []frontend.Variable {
    return u521Field.Reduce(v.Element).Limbs
}
```

Remediation

This issue has been acknowledged by Brevis, and a fix was implemented in commit ea757d48 ↗.

3.33. Number of limbs of `Uint521` not enforced

Target	brevis-sdk		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The sdk's `Uint521` type uses gnark's standard library implementation for emulated fields, in a configuration in which elements are stored in six limbs of 96 bits each:

```
var u521Field *emulated.Field[Uint521Field]

type Uint521Field struct{}

func (f Uint521Field) NbLimbs() uint { return 6 }

func (f Uint521Field) BitsPerLimb() uint { return 96 }

type Uint521 struct {
    *emulated.Element[Uint521Field]
}
```

However, the `Values` and `FromValues` implementations do not verify the number of limbs. Thus, `FromValues` accepts arbitrary numbers of arguments, using them as limbs, even less or more than six, and `Values` similarly does not check how many limbs it is returning:

```
func (v Uint521) Values() []frontend.Variable {
    u521Field.Reduce(v.Element)
    return v.Limbs
}

func (v Uint521) FromValues(vs ...frontend.Variable) CircuitVariable {
    n := emulated.ValueOf[Uint521Field](0)
    n.Limbs = vs
    return newU521(&n)
}
```

Based on a cursory look into gnark's `emulated.Field` implementation as used by `Uint521`, it appears that a variable number of limbs generally seems to be supported.

However, lack of limb-number checks means that `Values` does not always return a list of the same length. This can cause problems for example when attempting to use `Uint521` as components of `List` or `TupleN` types; see Discussion point 4.7.

Impact

As a constant length of the return value of `Values` is not ensured, using `Uint521` within the SDK's list or tuple types may cause crashes or panics or lead to unintended behavior (e.g., limbs of three `Uint521`s being split up among two `Uint521`s instead).

While it appears as though `emulated.Field` can handle elements with nonstandard numbers of limbs, we have not fully checked `emulated.Field` to make sure this is the case. As usage with non-standard number of limbs is not explicitly documented as being supported, it may thus be safer to ensure a standard number of limbs for `Uint521`.

Recommendations

We recommend to check that `len(vs)` is equal to `NumVars()` in `FromValues` and that `len(v.Limbs)` is equal to `NumVars()` just before the return in `Values`.

Remediation

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

- `11e19b8c`
- `494ff640`

3.34. Inconsistency in behavior of type conversions

Target	brevis-sdk		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The sdk provides a number of different types. To convert between them, sdk/circuit_api.go provides several conversion functions. Converting between types often involves choices; if the target type cannot represent parts of the range of the source type, should conversion fail for values outside of the target range, or should the source value be truncated and if so, how?

The following is the implementation of the `ToUint248` function:

```
// ToUint248 casts the input to a Uint248 type. Supports Uint32, Uint64,
    Uint248, Int248,
// Bytes32, and Uint521
func (api *CircuitAPI) ToUint248(i interface{}) Uint248 {
    switch v := i.(type) {
    case Uint248:
        return v
    case Int248:
        return newU248(v.Val)
    case Uint32:
        return newU248(v.Val)
    case Uint64:
        return newU248(v.Val)
    case Bytes32:
        api.g.AssertIsEqual(v.Val[1], 0)
        return newU248(v.Val[0])
    case Uint521:
        max248 := ConstUint521(MaxUint248)
        api.Uint521.AssertIsLessOrEqual(v, max248)
        bits := api.Uint521.ToBinary(v, numBitsPerVar)
        return api.Uint248.FromBinary(bits[:numBitsPerVar]...)
    }
    panic(fmt.Errorf("unsupported casting from %T to Uint248", i))
}
```

As seen here, for the source type `Bytes32`, the choice is to enforce that the source value was in the representable range; the 248 bits of the `Bytes32` that are stored in `Val[0]` are used for the `Uint248`,

and it is asserted that the remaining eight bits in `Val[1]` are all zero. Similarly, in the case of source type `Uint521`, it is asserted that the value is in the range representable in 248 bits.

In contrast, for source type `Int248`, the underlying value is used directly. This results in a wraparound modulo 2^{248}, which changes negative values. An alternative, which would be more in line with how the previously mentioned two types are treated, would be to assert that the original `Int248` was not negative.

The analogous behavior occurs in the `ToInt248` function when converting the other way:

```
func (api *CircuitAPI) ToInt248(i interface{}) Int248 {
    switch v := i.(type) {
    // ...
    case Uint248:
        return newI248(v.Val)
    }
    // ...
}
```

Here, values that are 2^{247} or bigger will be reinterpreted by subtracting 2^{248}. An alternative behavior would be to range check the value to be smaller than 2^{247}.

Impact

The behavior of type-conversion functions when the source value is not representable in the target value is not documented, with the implementation making inconsistent choices depending on the type. This risks users making incorrect assumptions about the behavior of these functions.

Recommendations

We recommend to document the behavior of the various type conversions. It could also be considered whether the conversions pointed out above should assert the source value to be in the range representable by the target type.

Remediation

This issue has been acknowledged by Brevis. In `acf1510f` ↗, documentations comments were added to clarify the implemented behavior of conversion between `Uint248` and `Int248`.

3.35. Low bit widths for `Transaction` fields

Target	brevis-sdk		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The `Transaction` struct is defined in sdk/circuit_input.go as follows:

```
type Transaction struct {
    ChainId Uint248
    BlockNum Uint32
    Nonce Uint248
    // GasTipCapOrGasPrice is GasPrice for legacy tx (type 0) and GasTipCapOap
    for
    // dynamic-fee tx (type 2)
    GasTipCapOrGasPrice Uint248
    // GasFeeCap is always 0 for legacy tx
    GasFeeCap Uint248
    GasLimit Uint248
    From Uint248
    To Uint248
    Value Bytes32
}
```

It is intended to reflect the data associated to an Ethereum transaction. The Ethereum yellow paper ↗ (version 9fde3f4 from September 2nd, 2024) specifies the fields associated to a transaction in section 4.2. The types of most of the fields of the above struct are specified in display (18) of the yellow paper, as follows:

- `Nonce` — 256-bit unsigned integer
- `GasTipCapOrGasPrice` — 256-bit unsigned integer
- `GasFeeCap` — 256-bit unsigned integer
- `GasLimit` — 256-bit unsigned integer
- `To` — 20-byte address or empty
- `Value` — 256-bit unsigned integer

Apart from `To` and `Value`, where the type in the struct above is sufficient to represent the entire range specified in the yellow paper, the other fields listed above are specified as 256-bit unsigned integers, but the struct only stores them as 248-bit unsigned integers. The gas-related values should in prac-

tice never become large enough to not fit into 248, though. The nonce is the number of transactions sent by the sender so far. It is not feasible to reach 2^{248} transactions on an account, so this value also fits in practice into 248 bits.

This leaves `From`, `ChainId`, and `BlockNum`. The block number is not a part of the transaction in the Ethereum specification, but transactions are part of a block with a block number, specified in section 4.4, and in display (44) as a natural number of arbitrary size. In practice, it will take more than 10^{68} years until the Ethereum block number exhausts 248 bits, so this field is not a problem either. The `From` field is not part of the transaction in the Ethereum specification but derived from the signature, but as addresses on Ethereum are 160 bits large, 248 bits suffice for `From` as well.

It remains to discuss `ChainId`. For the transaction, the Ethereum specification only specifies that the chain ID should be to the network's chain ID, which is 1 in the case of Ethereum Mainnet. EIP-155 ↗, which introduced the chain ID, did not specify a maximum size. EIP-1344 ↗ suggests that the chain ID is a 256-bit value. There was some discussion (see for example on GitHub here ↗ or on Ethereum Magicians here ↗) to restrict the range, but nothing passed by the time this report was written, as far as we are aware.

There are some suggestions that chain IDs may be generated as hashes, in which case chain IDs not fitting in 248 bits would be realistic. However, the actual range for the chain IDs in Brevis is restricted further. The `pack` function restricts `ChainId` to only 32 bits. There are already chains with chain IDs that are larger than 32 bits.

Impact

The current circuits will not be usable for transaction on chains with a chain ID equal to or bigger than 2^{32}.

Recommendations

Should Brevis wish to be usable for such chains, the bit width allowed for `ChainID` would need to be increased. Alternatively, we recommend to document as a comment in the code that it is an explicit choice.

Remediation

This issue has been acknowledged by Brevis. Brevis is planning to remove `ChainId` in a future iteration of Brevis SDK.

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. Various possible simplifications and optimizations

We collect here some simplifications and optimizations we encountered that are possible in the codebase.

More efficient implementation for function `Cmp`

In the sdk's sdk/utils.go, the function `Cmp` is implemented as follows[1]:

```
// Inspired by
// https://github.com/Consensys/gnark/blob/
// 429616e33c97ed21113dd87787c043e8fb43720c/frontend/cs/scs/api.go#L523
// To reduce constraints comsumption, use predefined number of variable's bits.
func Cmp(api frontend.API, i1, i2 frontend.Variable, nbBits int)
    frontend.Variable {
    bi1 := bits.ToBinary(api, i1, bits.WithNbDigits(nbBits))
    bi2 := bits.ToBinary(api, i2, bits.WithNbDigits(nbBits))

    var res frontend.Variable
    res = 0

    for i := nbBits - 1; i >= 0; i-- {
        iszeroi1 := api.IsZero(bi1[i])
        iszeroi2 := api.IsZero(bi2[i])

        i1i2 := api.And(bi1[i], iszeroi2)
        i2i1 := api.And(bi2[i], iszeroi1)

        n := api.Select(i2i1, -1, 0)
        m := api.Select(i1i2, 1, n)

        res = api.Select(api.IsZero(res), m, res)
    }
    return res
}
```

[1] The first line of the comment was broken into three lines to fit in the page width of this report.

This function checks that `i1` and `i2` are smaller than `2^nbBits`, and it constrains the return value to -1 if `i1 < i2`, to 0 if `i1 == i2`, and to 1 if `i1 > i2`.

The standard gnark function with the same purpose referred to in the comment works similarly, but it decomposes both `i1` and `i2` into as many bits as the bit width of the field over which the circuit variables are defined. In some cases (like in the context of the 32- and 64-bit types of the sdk, for which this function is used), it may be known that `i1` and `i2` should be representable with fewer bits, so it is more efficient in witnesses and constraints to use less, as implemented in the above function.

However, in those cases, there is a much more efficient way to implement this function. Let first `r` be the prime modulus of the field over which the circuit is defined, and `n` the bit width of `r` so that `2^n > r >= 2^(n-1)`. Assume that `a` and `b` are two values that satisfy `0 <= a, b < 2^k` for some `k` (in other words, `a` and `b` have at most bit width `k`). If `a < b`, we have `0 < b - a < 2^k` and `-2^k < a - b < 0`. Assuming `k <= n-2`, we get the following results: `0 < (b - a) % r < 2^k`, `r - 2^k < (a - b) % r < r`, and `2^k <= r - 2^k` (the latter is equivalent to `r >= 2^(k+1)`, and this holds as `k+1 <= n-1`). The upshot is that if we have constrained both `a` and `b` to be at most `k` bits wide, and `a - b` is also at most `k` bits wide, then we must have `a >= b`.

Checking that a circuit variable is at most `k` bits wide is more efficient in gnark than actually constructing the bit decomposition, and in the `Cmp` use case we do not actually need the bit decomposition. Thus, the following kind of implementation is more efficient:

```
func CmpNew(api frontend.API, i1, i2 frontend.Variable, nbBits int)
    frontend.Variable {
    if nbBits > api.Compiler().Field().BitLen()-2 {
        panic("CmpNew called with nbBits too large!")
    }

    rangeChecker := rangecheck.New(api)
    rangeChecker.Check(i1, nbBits)
    rangeChecker.Check(i2, nbBits)
    results, err := api.Compiler().NewHint(CmpHint, 1, i1, i2)
    result := results[0]
    if err != nil {
        panic(err)
    }

    // Enforce that result is -1, 0, or 1
    result_sq := api.Mul(result, result)
    api.AssertIsBoolean(result_sq)

    // 1 if i1 < i2 according to hint, 0 else
    first_smaller := api.IsZero(api.Add(result, 1))

    // Select which is the bigger/smaller of i1 and i2, according to the hint
    bigger := api.Select(first_smaller, i2, i1)
    smaller := api.Select(first_smaller, i1, i2)
```

```
    // Get the difference, which should be still nonnegative if the hint is
    correct.
    diff := api.Sub(bigger, smaller)
    rangeChecker.Check(diff, nbBits)

    // If hint said that i1 < i2 but in fact i2 > i1, then the above will fail,
    and vice versa.
    // So what is left is to distinguish between i1 == i2 and i1 != i2.
    equal := api.IsZero(diff)
    result_zero := api.IsZero(result)
    // equal and result_zero must be either both true or both false
    api.AssertIsEqual(equal, result_zero)

    return result
}

func CmpHint(_ *big.Int, inputs []*big.Int, results []*big.Int) error {
    results[0].SetInt64(int64(inputs[0].Cmp(inputs[1])))
    return nil
}
```

The new function performs a range check on both inputs to be at most `nbBits` wide, and then uses the comparison result from a hint to ensure the right difference `i1 - i2` or `i2 - i1` is at most `nbBits` wide as well, with some additional logic for handling distinguishing `i1 == i2` from `i1 != i2`. The above is what we wrote down quickly to demonstrate this, so some improvements may be possible.

We compared the two implementations with this small circuit:

```
const bitnum = 64

// Here we define which circuit variables our circuit will have.
type ExampleCircuit struct {
    A frontend.Variable // Private witness
    B frontend.Variable // Private witness
}
// This function defines the constraints for our circuit
func (circuit *ExampleCircuit) Define(api frontend.API) error {
    //CmpOld(api, circuit.A, circuit.B, bitnum)
    CmpNew(api, circuit.A, circuit.B, bitnum)
    //api.AssertIsDifferent(cmp_result, 2)
    //api.AssertIsEqual(cmp_result_old, cmp_result_new)
    return nil
}
```

For eight-bit–wide values, the old implementation takes 139 constraints, the new one 90. For 248-

bit–wide values, the difference is much more, the old implementation requiring 4,459 and the new only 904, reducing the number of constraints by about 80%.

The current, less efficient implementation of `Cmp` is currently used for the SDK's `Uint32` (571 vs 198 constraints, 65% saved) and `Uint64` (1,147 vs 318 constraints, 72% saved) types. For the `Uint248` type, the standard gnark implementation of `Cmp` is used. Here the standard gnark `Cmp` takes 5,381 constraints, whereas our `CmpNew` takes 904, saving 83%.

More efficient implementation for function `ABS`

In the sdk repo's sdk/api_int248.go, the `ABS` function for the `Uint248` type provided by the SDK is implemented as follows:

```
// ABS returns the absolute value of a
func (api *Int248API) ABS(a Int248) Uint248 {
    bs := api.ToBinary(a)
    signBit := bs[247] // ToBinary returns little-endian bits, the last bit is
    sign
    flipped := make([]frontend.Variable, len(bs))
    for i, v := range bs {
        flipped[i] = api.g.IsZero(v.Val)
    }
    absWhenOrigIsNeg := api.g.Add(1, api.g.FromBinary(flipped...))
    abs := api.g.Select(signBit.Val, absWhenOrigIsNeg, a.Val)
    return newU248(abs)
}
```

This implementation is correct but very inefficient. Converting a 248-bit circuit variable into the individual bits takes a significant number of additional circuit variables as well as constraints. From our testing, this function costs around 1,200 constraints. However, it is possible to implement it with much fewer constraints by not using bitwise logic. The following implementation only costs four constraints if the sign bit is cached, an improvement by a factor of over 300.

```
// ABS returns the absolute value of a
func (api *Int248API) ABSNew(a Int248) Uint248 {
    a = api.ensureSignBit(a)

    resultIfNonNeg := a.Val
    resultIfNeg := api.g.Sub(new(big.Int).Lsh(big.NewInt(1), 248), a.Val)
    result := api.g.Select(a.SignBit, resultIfNeg, resultIfNonNeg)
    return newU248(result)
}
```

It works by computing the correct result in the negative case directly from the value, by subtracting it from 2^{248}. Then, `select` is used similarly to the current implementation.

Unnecessary branch in `Pad101Bytes`

The function `Pad101Bytes` in the file zk-hash/keccak/periphery.go has an unnecessary branch. It is intended to implement the `pad10*1` padding function for keccak, and it is implemented as follows:

```
func Pad101Bytes(data []byte) []byte {
    miss := 136 - len(data)%136
    if len(data)%136 == 0 {
        miss = 136
    }
    data = append(data, 1)
    for i := 0; i < miss-1; i++ {
        data = append(data, 0)
    }
    data[len(data)-1] ^= 0x80
    return data
}
```

The if case does not actually do anything, however, as when `len(data)%136 == 0`, then `miss` is already 136. This branch can thus be removed.

Code duplication between `IsGreaterThan` and `IsLessThan`

Several types provided by the sdk offer `IsGreaterThan` and `IsLessThan` functions. These functions tend to be implemented independently but in a very parallel manner. Code duplication can be reduced by replacing the implementation of one of the two functions by a call to the other. For example, in the case of `Int248`, the implementation of `IsGreaterThan` is as follows, with `IsLessThan` implemented very similarly.

```
func (api *Int248API) IsGreaterThan(a, b Int248) Uint248 {
    a = api.ensureSignBit(a)
    b = api.ensureSignBit(b)

    cmp := api.g.Cmp(a.Val, b.Val)
    isGtAsUint := api.g.IsZero(api.g.Sub(cmp, 1))

    isLt := api.g.Lookup2(
        a.SignBit, b.SignBit,
        isGtAsUint, // a, b both pos
        0, // a neg, b pos
        1, // a pos, b neg
        isGtAsUint, // a, b both neg
    )

    return newU248(isLt)
```

```
}
```

This function could be replaced by the following implementation:

```
func (api *Int248API) IsGreaterThan(a, b Int248) Uint248 {
    return api.IsLessThan(b, a)
}
```

Code duplication in `ToBytes32`

In the sdk repository's sdk/circuit_api.go, the function `ToBytes32` constructs the limbs of the resulting `Bytes32` directly:

```
case Uint521:
    api.Uint521.AssertIsLessOrEqual(v, MaxBytes32)
    bits := api.Uint521.ToBinary(v, 32*8)
    lo := api.Uint248.FromBinary(bits[:numBitsPerVar]...)
    hi := api.Uint248.FromBinary(bits[numBitsPerVar:256]...)
    return Bytes32{Val: [2]variable{lo.Val, hi.Val}}
```

Instead, the `FromBinary` function for `Bytes32` could be used.

Unnecessary constraint in `Filter`

The function `Filter` in the sdk repository's sdk/datastream.go contains the following lines:

```
valid := ds.api.isEqual(ds.toggles[i], 1)
newToggles[i] = api.Select(api.And(toggle, valid), 1, 0)
```

Here, `ds.toggles` are intended to be Boolean. Generally, it appears to be the assumption that the caller will have constrained `ds.toggles[i]` to be Boolean already. In that case, the `isEqual` constraints are not necessary, and it would be equivalent to do `valid := ds.toggles[i]`.

However, even if we assume that `ds.toggles[i]` has not yet been constrained to be Boolean, the `isEqual` constraints are not necessary. This is because `api.And` will constrain both arguments to be Boolean as well. So the above two lines may be replaced by

```
newToggles[i] = api.Select(api.And(toggle, ds.toggles[i]), 1, 0)
```

Going further, `api.Select` will return `1` in this case if the selector is true (encoded by `1`), otherwise `0`. But as `api.And` will return a Boolean, so a value that is either `1` or `0`, the `api.Select` will just return

the selector. Hence, the above can be further reduced to

```
newToggles[i] = api.And(toggle, ds.toggles[i])
```

Note that `api.And` constraining both arguments to be Boolean is a not fully documented feature of the current gnark implementation. The API documentation says

```
// Or returns a & b
// a and b must be 0 or 1
And(a, b Variable) Variable
```

which could be read to suggest the caller must ensure `a` and `b` are 0 or 1.

However, the two constraint systems implement `And` as follows:

```
// Or returns a & b
// a and b must be 0 or 1
func (builder *builder) And(a, b frontend.Variable) frontend.Variable {
    builder.AssertIsBoolean(a)
    builder.AssertIsBoolean(b)
    res := builder.Mul(a, b)
    builder.MarkBoolean(res)
    return res
}
```

```
// And compute the AND between two frontend.Variables
func (builder *builder) And(_a, _b frontend.Variable) frontend.Variable {
    vars, _ := builder.toVariables(_a, _b)

    a := vars[0]
    b := vars[1]

    builder.AssertIsBoolean(a)
    builder.AssertIsBoolean(b)

    res := builder.Mul(a, b)
    builder.MarkBoolean(res)

    return res
}
```

In both cases, the two arguments are constrained to be Boolean. The above snippets are from github.com/celer-network/gnark@v0.1.0/frontend/cs/scs/api.go and github.com/celer-network/gnark@v0.1.0/frontend/cs/r1cs/api.go.

Dead code in `DefaultHostCircuit`

In the sdk repository's sdk/host_circuit.go, the function `DefaultHostCircuit` is implemented as follows:

```
func DefaultHostCircuit(app AppCircuit) *HostCircuit {
    maxReceipts, maxStorage, maxTxs := app.Allocate()
    var inputCommits = make([]frontend.Variable, NumMaxDataPoints)
    for i := 0; i < NumMaxDataPoints; i++ {
        inputCommits[i] = 0
    }
    h := &HostCircuit{
        Input: defaultCircuitInput(maxReceipts, maxStorage, maxTxs),
        Guest: app,
    }
    return h
}
```

Note that the part

```
var inputCommits = make([]frontend.Variable, NumMaxDataPoints)
for i := 0; i < NumMaxDataPoints; i++ {
    inputCommits[i] = 0
}
```

is actually never used and can be deleted.

Constraints added in `assertInputUniqueness` even when unneeded

In the sdk repository, the `assertInputUniqueness` function in sdk/host_circuit.go has a `shouldCheck` argument that is not a circuit variable but a native `int`, indicating whether the uniqueness check should be performed or not. However, constraints are added in either case:

```
func assertInputUniqueness(api frontend.API, in []frontend.Variable,
    shouldCheck int) {
    multicommit.WithCommitment(api, func(api frontend.API, gamma
    frontend.Variable) error {

        // [Some constraints added]

        for i := 0; i < len(sorted)-1; i++ {

            // [More constraints added]
```

```
            isValid = api.Select(shouldCheck, isValid, 1)
            api.AssertIsEqual(isValid, 1)
        }
        return nil
    }, in...)
}
```

A prover can always satisfy the constraints that were omitted in the above snippet. What a prover cannot always arrange is that `isValid` is 1 before the reassignment shown above. However, if `shouldCheck` is 0, then `isValid` will be 1 after the reassignment, no matter what it was before. So in that case, the prover can always satisfy the constraints introduced by this function. The constraints and associated witnesses introduced by `assertInputUniqueness` could thus be removed with no change to the validity of the statement being proven by the overall circuit. The function could thus be rearranged as follows:

```
func assertInputUniqueness(api frontend.API, in []frontend.Variable,
    shouldCheck int) {
    if shouldCheck == 1 {
        multicommit.WithCommitment(api, func(api frontend.API, gamma
    frontend.Variable) error {

            // [Some constraints added]

            for i := 0; i < len(sorted)-1; i++ {

                // [More constraints added]

                api.AssertIsEqual(isValid, 1)
            }
            return nil
        }, in...)
    }
}
```

This would save a significant number of constraints and witnesses (all that are introduced by this function) in the case that `shouldCheck` is not `1`. Additionally, in the case that `shouldCheck` is `1`, this will save the constraint and witness introduced by `api.Select(shouldCheck, isValid, 1)`.

Smaller multiplex in Keccak256

The `Keccak256` function in keccak/keccak256.go of the zk-hash repository uses a multiplex at the end of the function to select the final result:

```
selected := mux.Multiplex(api, roundIndex, 25, MAX_ROUNDS,
    transpose(outputStates))
```

```
for i := 0; i < 4; i++ {
    out[i] = selected[i]
}
```

Here, the multiplex is done first over the full 25-word state, and then only the first four words are copied to the output. It should be more efficient to just do the multiplex over the first four words. This would be similar to how it is done in `Keccak256Bits` in keccak/keccak256_bits.go, where the end of the function looks like this:

```
selected := mux.Multiplex(api, roundIndex, 256, maxRounds,
    transpose2(states[1:]))
copy(out[:], selected[:256])
```

The implementation of multiplex allows calling it with a lower number of output components (in this case, 256) than the input components (in this case, 1,600). The same pattern could be used in `Keccak256` in keccak256.go to save some constraints.

Simplified `Flip`

The function `Flip` in the zk-hash repository's utils/slice.go is used to reverse a slice. It is implemented as follows:

```
func Flip[T any](in []T) []T {
    res := make([]T, len(in))
    copy(res, in)
    for i := 0; i < len(in)/2; i++ {
        tmp := res[i]
        res[i] = res[len(res)-1-i]
        res[len(res)-1-i] = tmp
    }
    return res
}
```

Instead of reading the values to swap from `res`, they could be read from `in`, which removes the need to use the variable `tmp` and to copy `in` to `res` initially. However, to ensure that the middle component is populated correctly, the condition in the for loop should then be changed to `i < (len(in) + 1) / 2`:

```
func Flip[T any](in []T) []T {
    res := make([]T, len(in))
    for i := 0; i < (len(in) + 1) / 2; i++ {
        res[i] = in[len(res)-1-i]
        res[len(res)-1-i] = in[i]
    }
```

```
    return res
}
```

Redundant Boolean check in `decode`

The zk-hash helper function `decode`, defined in mux/multiplexer.go, is implemented as follows:

```
// decodes the input selector num into a bit mask
// e.g. width 8 select 3 -> 00010000
func decode(api frontend.API, width int, input frontend.Variable) (output
    []frontend.Variable, outputSuccess frontend.Variable) {
    outputSuccess = 0
    for i := 0; i < width; i++ {
        value := isEqual(api, i, input)
        output = append(output, value)
        outputSuccess = api.Add(outputSuccess, value)
    }
    api.AssertIsBoolean(outputSuccess)
    return
}
```

In the loop, `value` will be `0` unless `i` is equal to `input` modulo the field modulus `r` over which the circuit is defined. As `i` ranges from `0` to `width - 1`, and this function cannot feasibly be instantiated with `width` larger than the field modulus (a 254-bit prime), we can conclude that there will be at most a single value `i = input % r` for which `value` will be nonzero, where it will be `1`. It follows that `outputSuccess` must be either 0 or 1, so Boolean. The constraint introduced by `api.AssertIsBoolean(outputSuccess)` is thus redundant. The private function `decode` is also only used by `Multiplex` in the same file, where `outputSuccess` is constrained to be equal to 1 anyway.

4.2. Additional validation

We recommend to check the assumptions made in some functions for best practice. We collect such cases in this section.

Length of argument to `Hash2FV`

In the zk-bridge file circuits/fabric/headers/headerutil/utils.go, the function `Hash2FV` is implemented as follows:

```
func Hash2FV(h []byte) [2]frontend.Variable {
    return [2]frontend.Variable{
        new(big.Int).SetBytes(h[:16]),
        new(big.Int).SetBytes(h[16:]),
    }
}
```

This function is used to split up 32 bytes into two circuit variables. However, this function does not check that the input is actually 32 bytes long. Should the input be less than 16 bytes long, the code will panic due to an out-of-bounds access. In other cases, in which callers provide an input of length unequal to 32, behavior may be different than intended but not result in an error being returned or a runtime panic. We recommend to verify the length of `h` before use.

Size and range checks for decompose-related functions

The function `decompose`, which is implemented as displayed below, can lead to unintended behavior when `bitSize` is larger than the minimum of 64 and the number of bits that can be stored in type `T`.

```
func decompose[T uint | byte](data *big.Int, bitSize uint, length uint) []T {
    res := decomposeBig(data, bitSize, length)
    ret := make([]T, length)
    for i, limb := range res {
        ret[i] = T(limb.Uint64())
    }
    return ret
}
```

We recommend to document that this function should be used carefully regarding `T`'s bit size, or add an assertion checking if `bitSize` is a valid size.

Also, the function `decomposeBitsExact` extracts the limbs of the absolute value of the input, which cannot be inferred from the function's name itself. We suggest changing the function's name to more detailed one like `decomposeBitsExactOfAbs`, or specifically document this behavior of the function.

Padded length-size check for function `padBitsRight`

The following is the implementation of the function `padBitsRight` in the sdk repository's sdk/utils.go.

```
func padBitsRight(bits []uint, n int, with uint) []uint {
    ret := make([]uint, n)
    for i := 0; i < len(bits); i++ {
```

```
        ret[i] = bits[i]
    }
    for i := len(bits); i < n; i++ {
        ret[i] = with
    }
    return ret
}
```

If `len(bits) > n`, so that the function should pad `bits` to length `n`, but `bits` is already longer than that, then the function will panic due to an out-of-bounds write. It may be neater to check if `len(bits) > n` and panic with a more descriptive error message, though that extra check will of course make the function slightly slower, so it is a trade-off.

Function `FromBinary` number or arguments

The following is the implementation of the `FromBinary` function for the sdk's `Bytes32` type, from sdk/api_bytes32.go:

```
// FromBinary interprets the input vs as a list of little-endian binary digits
// and recomposes it to a Bytes32. Input size can be less than 256 bits, the
// input is padded on the MSB end with 0s.
func (api *Bytes32API) FromBinary(vs ...Uint248) Bytes32 {
    var list List[Uint248] = vs
    values := list.Values()
    for i := len(vs); i < 256; i++ {
        values = append(values, 0)
    }
    res := Bytes32{}
    res.Val[0] = api.g.FromBinary(values[:numBitsPerVar]...)
    res.Val[1] = api.g.FromBinary(values[numBitsPerVar:]...)
    return res
}
```

The arguments `vs` should be at most 256 many; however, there are no checks being done to ensure this, so passing more than 256 bits will just pass that through and can make `res.Val[1]` more bits wide than intended. Then, if `ToBinary` is called on this `Bytes32` object, the constraints will not be satisfiable due to `Val[1]` not being eight bits wide only. We suggest making `FromBinary` already panic with a descriptive error message whenever `len(vs) > 256`. This will aid in debugging for users.

The same situation is present for the other types' `FromBinary` function.

Check for number of bits for `Uint521` function `ToBinary`

The sdk's `Uint521` type has a `ToBinary` function that is implemented as follows in sdk/api_uint521.go:

```
// ToBinary decomposes the input v to a list (size n) of little-endian binary
    digits
func (api *Uint521API) ToBinary(v Uint521, n int) List[Uint248] {
    reduced := api.f.Reduce(v.Element)
    bits := api.f.ToBits(reduced)
    ret := make([]Uint248, n)
    for i := 0; i < n; i++ {
        ret[i] = newU248(bits[i])
    }
    return ret
}
```

Should `n > len(bits)`, a panic will occur due to out-of-bounds access in the loop. We recommend to instead explicitly check for `n > len(bits)` or `n > 521` and panic with a descriptive error message in that case.

Additional length check regarding conversion between bits and bytes

Function `bits2Bytes` in the sdk repository's sdk/host_circuit.go and function `addOutput` in sdk/circuit_api.go are implemented as the following:

```
func (api *CircuitAPI) addOutput(bits []variable) {
    // the decomposed v bits are little-endian bits. The way evm uses Keccak
    expects
    // the input to be big-endian bytes, but the bits in each byte are little
    endian
    b := flipByGroups(bits, 8)
    api.output = append(api.output, b...)
    dryRunOutput = append(dryRunOutput, bits2Bytes(b)...)
}
```

```
func bits2Bytes(data []frontend.Variable) []byte {
    var bits []uint
    for _, b := range data {
        bits = append(bits, uint(fromInterface(b).Int64()))
    }

    bytes := make([]byte, len(bits)/8)
    for i := 0; i < len(bits)/8; i++ {
        for j := 0; j < 8; j++ {
```

```
            bytes[i] += byte(bits[i*8+j] << j)
        }
    }

    return bytes
}
```

Both functions relate to converting bits into bytes and are implemented in a way that will lead to incorrect results should the input length not be divisible by 8. We thus recommend to check this and panic otherwise.

Length of `FlipByGroups` being a multiple of `groupSize`

The function `FlipByGroups` in the zk-hash repository's utils/binary.go is implemented as following.

```
// FlipByGroups flips the order of the groups of groupSize. e.g. [1,2,3,4,5,6]
    with groupSize 2 is flipped to [5,6,3,4,1,2]
func FlipByGroups[T any](in []T, groupSize int) []T {
    res := make([]T, len(in))
    copy(res, in)
    for i := 0; i < len(res)/groupSize/2; i++ {
        for j := 0; j < groupSize; j++ {
            a := i*groupSize + j
            b := len(res) - (i+1)*groupSize + j
            res[a], res[b] = res[b], res[a]
        }
    }
    return res
}
```

The length of `in` should be a multiple of `groupSize` for this implementation to work properly; otherwise, the result would be unexpected. However, the current implementation does not check the length, and the following unexpected behavior is possible.

```
in: [1,2,3,4,5,6,7], groupSize: 2
res: [6,7,3,4,5,1,2]
```

We suggest adding a check for `len(in) % groupSize == 0`.

The sdk function `OutputUint` does not check maximum bit size

The following is the documentation and implementation for the function `OutputUint` in sdk/circuit_api.go of the sdk repository:

```
// OutputUint adds an output of solidity uint_bitSize type where N is in range
    [8, 248]
// with a step size 8. e.g. uint8, uint16, ..., uint248.
// Panics if a bitSize of non-multiple of 8 is used.
// Panics if the bitSize exceeds 248. For outputting uint256, use
    OutputBytes32 instead
func (api *CircuitAPI) OutputUint(bitSize int, v Uint248) {
    if bitSize%8 != 0 {
        panic("bitSize must be multiple of 8")
    }
    b := api.g.ToBinary(v.Val, bitSize)
    api.addOutput(b)
    _, ok := v.Val.(*big.Int)
    dbgPrint(ok, "added uint%d output: %d\n", bitSize, v.Val)
}
```

It is said that the function panics if `bitSize` is not a multiple of 8 or exceeds 248. This is a good sanity check to prevent incorrect usage. However, the implementation does not actually panic when `bitSize` exceeds 248. We recommend to add that check.

Negative offsets in `offsetSlot`

The `offsetSlot` function in the sdk's sdk/circuit_api.go is passed a slot and an offset, and it is roughly intended to return their sum. The function's implementation begins as follows:

```
func (api *CircuitAPI) offsetSlot(slotBits [256]variable, offset int)
    [256]variable {
    if offset <= 0 {
        return slotBits
    }
```

For `offset = 0`, the function returns `slotBits`, as is reasonable. However, for negative values of `offset`, the function also returns `slotBits`. If a negative offset should be supported, then it would make more sense to return the sum, as for the positive offsets. If negative offsets are not intended to be supported, then the function should reject them by panicking with a descriptive error message instead.

Number of values in `Reduce`

In the sdk repository's sdk/datastream.go, the function `Reduce` is implemented as follows:

```
func Reduce[T, R CircuitVariable](ds *DataStream[T], initial R, reducer
    ReduceFunc[T, R]) R {
```

```
    acc := initial
    for i, data := range ds.underlying {
        newAcc := reducer(acc, data)
        oldAccVals := acc.Values()
        values := make([]frontend.Variable, len(oldAccVals))
        for j, newAccV := range newAcc.Values() {
            values[j] = ds.api.g.Select(ds.toggles[i], newAccV, oldAccVals[j])
        }
        acc = acc.FromValues(values...).(R)
    }
    return acc
}
```

Note that, should `newAcc.Values()` be of a different length than `acc.Values()` in the loop body, then the function would not behave as intended. If the length is smaller, then `values` will be padded with default values at the end, which may cause the new `acc` to be incorrect. If the length is bigger, then the inner loop will attempt to make an assignment out of bounds, causing a panic.

The `Reduce` function should not be called with arguments for which such inconsistent lengths are possible, so this could be considered a user error. However, to prevent incorrect use and make debugging easier, we recommend to check that `oldAccVals` and `newAcc.Values()` have the same number of elements and, if they do not, the function should panic.

Checking for pinned data at too-high index

The `BuildCircuitInput` function in the sdk's sdk/app.go is used to, in particular, convert from `rawData` (for receipt, storage, and transaction data) to `CircuitInput`. See the descriptions in Findings 3.22. ↗ and 3.23. ↗ for a short overview of `rawData`. As the three types of data are handled completely analogously, we will in the following only mention receipts, but everything applies also to storage and transaction data.

To check that the `rawData` is compatible with the maximum sizes allocated for the `CircuitInput`, the function `checkAllocations` is used. For receipts, it carries out the following checks:

```
numReceipts := len(q.receipts.special) + len(q.receipts.ordered)
if maxReceipts%32 != 0 {
    return allocationMultipleErr("receipt", maxReceipts)
}
if numReceipts > maxReceipts {
    return allocationLenErr("receipt", numReceipts, maxReceipts)
}

// ..

total := maxReceipts + maxSlots + maxTxs
if total > NumMaxDataPoints {
```

```
    return allocationLenErr("total", total, NumMaxDataPoints)
}
```

The actual data to be assigned is only checked with the second of the two receipt-specific tests. What is checked is whether the number of pinned and unpinned data points is at most the maximum number of receipts. However, this does not ensure that all receipts will be assigned with indexes from `0` to `maxReceipts - 1`, as a receipt may have been pinned at a larger index. The remaining check `total > NumMaxDataPoints` only involves checking the total number of pinned or unpinned receipts, storage, and transaction data, which will also not catch a too-high pinned index.

After using `checkAllocations`, the `BuildCircuitInput` will call `assignReceipts` to actually assign the `CircuitInput`. Should a pinned index be too high, an out-of-bounds assignment will cause a panic in that function.

So the mistake of pinning at a too-large index will be caught. However, it would be easier to debug if this were checked in `checkAllocations`, by calculating the maximum key of `q.receipts.special` and returning an error with a descriptive error message if that value is bigger than or equal to `maxReceipts` — similarly for storage and transactions.

Possible out-of-bounds access in `dotProduct`

In the zk-hash repository, in mux/multiplexer.go, the `dotProduct` function would crash if called with a too-large value for `width`, due to out-of-bounds access. We recommend to check whether `width > len(inputA)` and panic with a descriptive error message if that holds.

Additional asserts for `Pad101Bits`

In the zk-hash implementation of the `pad10*1` padding for Keccak in keccak/pad.go, the `Pad101Bits` function assumes that `inBits` divides 8. We recommend to check this and panic with a descriptive error message if it is not satisfied, to help avoid incorrect usage.

Additionally, we recommend to panic with an error message if `inLenMin > inLenMax`.

Checking for consistent column lengths in `transpose`

The zk-hash implementation of Keccak contains in keccak/keccak256.go an implementation of transposition (so switching of rows and columns):

```
func transpose(input [][]frontend.Variable) [][]frontend.Variable {
    rows := len(input)
    cols := len(input[0])
    output := make([][]frontend.Variable, cols)

    for i := 0; i < cols; i++ {
```

```
        output[i] = make([]frontend.Variable, rows)
        for j := 0; j < rows; j++ {
            output[i][j] = input[j][i]
        }
    }
    return output
}
```

Note that the number of columns is obtained from the length of the first row, `input[0]`. Because of this, should there be a `j` with `len(input[j]) > len(input[0])`, the extra components would not be copied to the output. If `len(input[j]) < len(input[0])`, then a panic due to an access out of bounds would happen when attempting to access `nput[j][i]` in the inner loop.

This is fine with regards to the intended functionality, as this function should only be called with rectangle-shaped inputs. However, we recommend to check that this is the case to make it easier to debug incorrect usage, by panicking if `len(input[i]) != cols` for any `1 <= i < rows`.

Size check for `Uint64s2Blocks`

In keccak/periphery.go in the zk-hash repository, it might make sense in the `Uint64s2Blocks` function to check and panic with an error message if `len(padded) > 17*MAX_ROUNDS`. Currently, this would cause an out-of-bounds array access.

4.3. Lack of documentation or comments

The code in scope generally lacks comments or documentation. We would recommend for best practice to document what individual functions, structs, and so forth are intended to do using documenting comments. This can help avoid mistakes while also improving developer experience.

In some cases, intended behavior of a function may be guessed based on its name; however, this is often not necessarily the case. As a concrete example, the function `byte2Nibs` in zk-bridge's circuits/fabric/headers/circuit.go can be guessed from its name to convert a byte into two nibbles. Actually, the function is passed not a byte that gets decomposed but bits, and "byte" refers to the requirement that the input be exactly eight bits long. Also, the function name alone leaves open with what endianess the input will be interpreted and in which order the nibbles will be returned. As it turns out, the input bits are interpreted as little endian, and the return nibbles are returned in big-endian order. This should be documented with a documentation comment above the function.

Below we collected some concrete instances where we recommend additional or changed comments or other documentation.

Validity of data not checked by some constructors of sdk types

Types offered by the sdk provide a `Values` method function that will provide the data underlying the object as a raw list of circuit variables (or circuit constants). Analogously, such lists of raw values can be used to set the type with the method function `FromValues`. These functions could thus be described as providing serialization and deserialization to these types.

On `FromValues`, the raw inputs are used as is and are not constrained to be well-formed in accordance to their use in the respective type. For example, for the `Uint64` type, the `Val` field should contain a value of bit width at most 64. However, `FromValues` does not enforce this:

```
func (v Uint64) FromValues(vs ...frontend.Variable) CircuitVariable {
    if len(vs) != 1 {
        panic("Uint64.FromValues only takes 1 param")
    }
    v.Val = vs[0]
    return v
}
```

In most use cases of `FromValues`, the data passed to it will be enforced to be well-formed already, so not checking this again in this deserialization function is in those cases unnecessary; avoiding these checks saves circuit variables and constraints. In other cases, the caller may need to ensure well-formedness of the input. We recommend to document this.

Similarly, most types offer internal functions with names beginning with `new`, such as `newU64`, to construct the type. These also tend to not validate the input. For example, `newU64` is implemented as follows:

```
func newU64(v frontend.Variable) Uint64 {
    return Uint64{Val: v}
}
```

We recommend to document this in a documentation comment to ensure developers using these functions are aware of this. In the current codebase, various functions such as `Uint64API.Mul` do use `newU64` in a way that could result in values that are larger than a bit width of at most 64; however, these functions do have documentation comments warning that an overflow may happen.

Consistency check for sign bit when using `newI248`

The `newI248` in sdk/api_int248.go is implemented as follows:

```
func newI248(v ...frontend.Variable) Int248 {
    ret := Int248{Val: v[0]}
    if len(v) > 1 {
        ret.SignBit = v[1]
```

```
            ret.signBitSet = true
        }
        return ret
    }
```

The first argument is stored as `Val` in the returned `Int248` and the second as the cached sign `SignBit`. However, no consistency check is done to ensure that the sign bit in `Val` matches the one cached in `SignBit`. Thus, the caller must ensure this. We recommend to document this clearly.

Dealing with native types where witnesses may be used as well

Witnesses in the circuit are assigned values in a finite field of prime order `r`. When values, for example of type `big.Int`, are used for assignments, they will thus be reduced modulo `r`. In many places, native types can be used as well instead of witnesses, essentially acting as constants from the perspective of the circuit. In those cases, native types are used as is and not reduced modulo `r`. In some situations, such constants can result in a different behavior than the corresponding witnesses; see for example Finding 3.20. ↗. Beyond the concrete issue of that finding, it may be considered to generally deal with potential differences in behavior in one of the following three ways:

1. Restrict the values accepted when instantiating constants by only allowing values `v` satisfying `0 <= v < r`.
2. Reduce the value modulo `r`.
3. Document any differing behavior.

Ambiguous documentation regarding overflows in the sdk's types

The documentation for various arithmetic operations for types provided by the sdk warn about overflows as in the following example:

```
// Add returns a + b. Overflow can happen if a + b > 2^248
func (api *Uint248API) Add(a, b Uint248, other ...Uint248) Uint248 {
```

The sdk is otherwise designed in such a way as to make its usage accessible to users not very familiar with how ZK circuits work. Such users might expect that the overflow that can happen for a `Uint248` type will in practice be reduction modulo 2^{248}. For example, they might expect that if they multiply by 2, this then corresponds to shifting bits to the left, with the most significant bit not being retained.

But this is not the case. Circuit variables store values that are elements of a finite field of prime order `r`, and arithmetic operations are performed modulo `r`. Functions such as the above contain no additional logic to change this. Thus, if the new value will be wider than (in the case of `Uint248`) 248 bits, then functions such as `ToBinary` might now not be satisfiable anymore on the result. Additionally, if the resulting value is large enough as an integer, a wraparound modulo a prime `r` will happen. The documentation should clarify this behavior.

Documentation comments for `OutputUint32` and `OutputUint64`

The documentation comments for `OutputUint32` and `OutputUint64` in sdk/circuit_api.go of the sdk repository are the same as for `OutputUint` and have not been adjusted.

Arguments of `SlotOfArrayElement`

The `SlotOfArrayElement` function in the sdk repository's sdk/circuit_api.go is implemented and documented as follows:

```
// SlotOfArrayElement computes the storage slot for an element in a solidity
// array state variable. arrSlot is the plain slot of the array variable.
// index determines the array index. offset determines the
// offset (in terms of bytes32) within each array element.
func (api *CircuitAPI) SlotOfArrayElement(arrSlot Bytes32, elementSize int,
    index, offset Uint248) Bytes32 {
    //api.Uint248.AssertIsLessOrEqual(offset, ConstUint248(elementSize))
    o := api.g.Mul(index.Val, elementSize)
    return Bytes32{Val: [2]variable{
        api.g.Add(arrSlot.Val[0], o, offset.Val),
        arrSlot.Val[1],
    }}
}
```

In Solidity's storage layout for dynamic arrays, the storage slot of a dynamic array itself contains the length of the array. The actual data of the array then begins at the address given by the Keccak hash of the address of the slot of the array itself. See the Solidity docs ↗ for more details.

In the function `SlotOfArrayElement` above, `arrSlot` should be the address of the slot at which the data of the array starts, not the address of the slot in which the length of the array is stored, so the caller must have already taken the Keccak hash. This is not made clear in the documenting comment and should be clarified, in particular as the related `SlotOfStructFieldInMapping` function *does* expect the slot of the state variable itself and takes Keccak itself (which cannot be avoided in that case, as the hash also includes the key).

Additionally, it should be clarified that `elementSize` is intended as the number of storage slots each element uses, not, for example, the size in bytes. Similarly, `offset` is also counted in slots, but this is already clarified by the current documentation comment.

Should a single element of the dynamic array require 16 or fewer bytes, then more than one element will be packed into the same slot. As `elementSize` must be a nonnegative integer, it is not possible to use `SlotOfArrayElement` to compute addresses in the array in such cases. It would be good to document this limitation as well.

Finally, the way the two limbs of the result are calculated does not account for potential overflows of the lower limb. Thus, the result will not be correct should the sum of the lower 248 bits of the array's initial address and `o + offset.Val` be bigger than or equal to 2^{248}. However, `o` and `offset.Val` will

usually be much smaller than 248 bits wide, so `arrSlot.Val[0]` would need to be very close to 2^{248} for this to happen. As normally this would be the lower 248 bits of a Keccak hash, this is extremely improbable to happen, so this should not be an issue in practice. It could be useful to document the reasoning as comments in or above the function to clarify that this edge case has been considered and the current implementation is a deliberate compromise.

Argument of `SlotOfStructFieldInMapping`

Similarly, the `offset` argument for the `SlotOfStructFieldInMapping` function in the sdk repository's sdk/circuit_api.go is described as `index of the struct value`. It would be useful to clarify that this is an offset counted in slots.

Sum instead of product symbol in `assertInputUniqueness` comment

In the function `assertInputUniqueness` in sdk/host_circuit.go of the sdk repository, there is the following comment:

```
// Grand product check. Asserts the following equation holds:
// ∑_{a \in in} a+ɣ = ∑_{b \in sorted} b+ɣ
```

Instead of ∑, it should be ∏.

Endianess in common/util/utils.go

The sdk repository's file common/util/utils.go contains functions `Recompose6BytesToNibbles`, `Recompose32ByteToNibbles`, and `RecomposeSDKByte32ToNibble`, for which we recommend to document what endianess arguments and return values have.

Success not constrained in `decode`

In the zk-hash repository, in mux/multiplexer.go, the `decode` function is used to convert a selector into a bitmask. It does not check that the selector is within the range of the bitmask width, so if the selector is out of range, the zero bitmask will be returned. However, the function returns a Boolean `outputSuccess` indicating whether the selector was in range or not. Thus, a caller can ensure that at least one output is selected by constraining `outputSuccess` to be 1, as is done by the `Multiplex` function. We recommend to explain this in the documentation comment of `decode`, as based on the current comment one may think that the function already constrains `input` this way.

Incomplete comment regarding Poseidon constants

The zk-hash file poseidon/circuit/constants.go constrains the following comment at the top regarding the parameters for the Poseidon hash function defined in that file:

```
// Parameters are generated by a reference script
    https://extgit.iaik.tugraz.at/krypto/hadeshash/-
    /blob/master/code/generate_parameters_grain.sage
// Used like so: sage generate_parameters_grain.sage 1 0 254 2 8 56
    0x30644e72e131a029b85045b68181585d2833e84879b9709143e1f593f0000001
```

This comment is not fully complete, however. While the parameters ultimately come from that script (though generate_parameters_grain.sage is deprecated, and generate_params_poseidon.sage should be used instead according to the authors), they are modified to work with the optimized algorithm used by the circuit. For example, the round constants apart from those for the first round are different than the ones generated by the script. See Discussion section 4.15. ↗ for a little more on this and an explanation for how we verified that the constants in constants.go are correct.

Assumption that 8 divides `inBits` is not documented for `Pad101Bits`

In the zk-hash implementation of the `pad10*1` padding for Keccak in keccak/pad.go, the `Pad101Bits` function assumes that `inBits` divides 8, but this is not documented.

Keccak functions require padded input

The zk-hash repository offers multiple functions to calculate the Keccak hash (`Keccak256BitsOptimized`, `Keccak256Bits`, and `Keccak256`). These all require input that has already been padded to a multiple of the block size of 1,088 bits with the `pad10*1` padding function, for which an implementation `Pad101Bits` is available in keccak/pad.go.

We recommend to document for these Keccak functions that they require padded input.

4.4. Minor recommendations

In this section we collect some minor recommendations.

`String` function for lists

The `String` function for lists just places a comma and space consecutively (", ") between each component. Adding, for example, the left square bracket "[" at the start and the right square bracket "]" at the end, similarly to how it is done for tuples with the left and right parentheses "(" and ")", would make it easier to understand the output.

Typo in one of the constants for Poseidon implementation in zk-hash

In the zk-hash repository's poseidon/circuit/constants.go, the very first string, in the function `strPOSEIDON_C` and for case `t == 2`, which should consist of a flat list, begins correctly with a single left square bracket "[" introducing the list, but it ends with two right square brackets, "]". Both of the bracket characters are replaced by an empty string in the function `parseOneDimensionArray` that is used to parse this string, which is why this does not cause an error on runtime.

```
    s := ""
    if t == 2 {
        s =

    `[0x9c46e9ec68e9bd4fe1faaba294cba38a71aa177534cdd1b6c7dc0dbd0abd7a7,
// ...
0xe3eca007699dd0f852eb22da642e495f67c988dd5bf0137676b16a31eab4667]
]`
```

We recommend to remove the last right square bracket, `]`.

Ambiguous genericity in `ToUint521`

The `ToUint521` in the brevis-sdk file sdk/circuit_api.go has the cases for `Bytes32` and `Uint248` implemented as follows:

```
case Bytes32:
    // Recompose the Bytes32 into BigField.NbLimbs limbs
    bits := v.toBinaryVars(api.g)
    f := Uint521Field{}
    limbs := make([]variable, f.NbLimbs())
    b := f.BitsPerLimb()
    limbs[0] = api.g.FromBinary(bits[:b]...)
    limbs[1] = api.g.FromBinary(bits[b : 2*b]...)
    limbs[2] = api.g.FromBinary(bits[2*b:]...)
    limbs[3], limbs[4], limbs[5] = 0, 0, 0
    el := api.Uint521.f.NewElement(limbs)
    return newU521(el)
case Uint248:
    el := api.Uint521.f.NewElement([]variable{v.Val, 0, 0, 0, 0, 0})
    return newU521(el)
```

Note that in the `Bytes32` case, the implementation starts in a way that does not hardcode the number of limbs of a `Uint521`, by instantiating `limbs` with `f.NbLimbs()` components instead of a hardcoded 6. However, the code following this assumes that this is what the value was; if `f.NbLimbs()` were smaller, one of the assignments would panic due to an out-of-bounds write attempt. In the second case shown, `Uint248` constructs the result, assuming six limbs as well.

Similarly, the function appears to be independent of the precise value of `f.BitsPerLimbs()` but makes the implicit assumption that `3*b >= 256` so that the 256 bits from `Bytes32` fits into the first three limbs.

We recommend to make the code more consistent by either hardcoding actual values throughout in this function or making the implementation independent of the values.

In the former case, it would be good to assert that the values used are equal to `f.NbLimbs()` and `f.BitsPerLimb()` to ensure that, should one or both change later, it will not be forgotten to update the `ToUint521` function.

String representation of `LogField` is empty

In the sdk repository's sdk/circuit_input.go, the type `LogField` and its method function `String` is defined. They are given by the following:

```
// LogField represents a single field of an event.
type LogField struct {
    // The contract from which the event is emitted
    Contract Uint248
    // The event ID of the event to which the field belong (aka topics[0])
    EventID Uint248
    // Whether the field is a topic (aka "indexed" as in solidity events)
    IsTopic Uint248
    // The index of the field. For example, if a field is the second topic of a
    log, then Index is 1; if a field is the
    // third field in the RLP decoded data, then Index is 2.
    Index Uint248
    // The value of the field in event, aka the actual thing we care about,
    only 32-byte fixed length values are supported.
    Value Bytes32
}

func (f LogField) String() string { return "" }
```

The string representation of a `LogField` is thus just the empty string. In contrast, similar types have nonempty string representations:

```
func (s StorageSlot) String() string { return "StorageSlot" }

func (r Receipt) String() string { return "Receipt" }

// ...
```

It may be considered to return a descriptive string for `LogField` as well (such as `"LogField"`).

Inconsistent serialization of `Transaction` struct

In the sdk repository, a `Transaction` struct is defined in sdk/circuit_input.go as follows:

```
type Transaction struct {
    ChainId Uint248
    BlockNum Uint32
    Nonce Uint248
    // GasTipCapOrGasPrice is GasPrice for legacy tx (type 0) and GasTipCapOap
    for
    // dynamic-fee tx (type 2)
    GasTipCapOrGasPrice Uint248
    // GasFeeCap is always 0 for legacy tx
    GasFeeCap Uint248
    GasLimit Uint248
    From Uint248
    To Uint248
    Value Bytes32
}
```

This struct is serialized with a `pack` function:

```
func (t Transaction) pack(api frontend.API) []variable {
    var bits []variable
    bits = append(bits, api.ToBinary(t.BlockNum.Val, 8*4)...)
    bits = append(bits, api.ToBinary(t.ChainId.Val, 8*4)...)
    bits = append(bits, api.ToBinary(t.Nonce.Val, 8*4)...)
    bits = append(bits, api.ToBinary(t.GasTipCapOrGasPrice.Val, 8*8)...)
    bits = append(bits, api.ToBinary(t.GasFeeCap.Val, 8*8)...)
    bits = append(bits, api.ToBinary(t.GasLimit.Val, 8*4)...)
    bits = append(bits, api.ToBinary(t.From.Val, 8*20)...)
    bits = append(bits, api.ToBinary(t.To.Val, 8*20)...)
    bits = append(bits, t.Value.toBinaryVars(api)...)
    return packBitsToFr(api, bits)
}
```

Note that in `pack`, the field `BlockNum` is serialized before `ChainId`, while in the struct these fields are listed in the other order. This inconsistency is not a problem in itself, but it could cause confusion later in development, so we recommend to keep the order consistent.

Incorrect error message in `assignToggleCommitment`

In the sdk repository's sdk/app.go, in `assignToggleCommitment`, there is the following snippet:

```
result, err := hasher.Sum()
if err != nil {
    panic(fmt.Sprintf("invalid toggles length %d", len(toggles)))
}
```

The error message was likely copied from another error earlier in the function and does not fit this error. We recommend to update the error message to something like the following:

```
panic(fmt.Sprintf("failed to hash toggle bit leaf: %s", err.Error()))
```

Incomplete or misleading error messages in `validateInput`

The `validateInput` function in the sdk/host_circuit.go file of the sdk repository carries out some sanity checks regarding the inputs. It is implemented as follows:

```
func (c *HostCircuit) validateInput() error {
    d := c.Input
    inputLen := len(d.Receipts.Raw) + len(d.StorageSlots.Raw)
    + len(d.Transactions.Raw)
    if inputLen > NumMaxDataPoints {
        return fmt.Errorf("input len must be less than %d", NumMaxDataPoints)
    }
    maxReceipts, maxSlots, maxTransactions := c.Guest.Allocate()
    if len(d.Receipts.Raw) != len(d.Receipts.Toggles) || len(d.Receipts.Raw)
    != maxReceipts {
        return fmt.Errorf("receipt input/toggle len mismatch: %d vs %d",
            len(d.Receipts.Raw), len(d.Receipts.Toggles))
    }
    if len(d.StorageSlots.Raw) != len(d.StorageSlots.Toggles)
    || len(d.StorageSlots.Raw) != maxSlots {
        return fmt.Errorf("storageSlots input/toggle len mismatch: %d vs %d",
            len(d.StorageSlots.Raw), len(d.StorageSlots.Toggles))
    }
    if len(d.Transactions.Raw) != len(d.Transactions.Toggles)
    || len(d.Transactions.Raw) != maxTransactions {
        return fmt.Errorf("transaction input/toggle len mismatch: %d vs %d",
            len(d.Transactions.Raw), len(d.Transactions.Toggles))
    }
    return nil
}
```

The first check is to ensure that the total number of inputs is not higher than the maximum that is allowed, `NumMaxDataPoints`. The error message `fmt.Errorf("input len must be less than %d",`

`NumMaxDataPoints)` is inaccurate as it suggests that an input length that is equal to the maximum is not allowed, while it is. Thus, `fmt.Errorf("input len must be less than or equal to %d", NumMaxDataPoints)` would be more accurate.

The other three checks are all analogous and check that for each of the three input types, the amount of raw data entries matches the configured allocation and that this is also equal to the number of toggles. The error messages, however, only mention the input and toggle lengths and suggest that they mismatch, when the error could also have arisen due to a mismatch with the configured maximum amount. Updating the first error message to

```
fmt.Errorf("mismatch between receipt input length, toggle length, or allocated
    receipt length: %d vs %d vs %d",
            len(d.Receipts.Raw), len(d.Receipts.Toggles), maxReceipts)
```

would make it more accurate and useful — analogously for the other two types.

Typo in `buildSendRequestCalldata` error message

The `buildSendRequestCalldata` function in sdk/app.go of the sdk repository has a typo in an error message, where `fonud` is used instead of `found`.

Typo in `calMerkelRoot` function name

The function `calMerkelRoot` in the sdk repository's sdk/host_circuit.go is used to calculate the root hash of a Merkle tree. The function name has a typo however, using "Merkel" instead of "Merkle".

Ethereum header `Nonce` not zero

In the zk-bridge repository's circuits/fabric/headers/util.go, the function `genHeader` generates an Ethereum header with dummy values. The field `Nonce` is filled with the byte 1:

```
Nonce: [8]byte{1, 1, 1, 1, 1, 1, 1, 1},
```

The Ethereum yellow paper describes the `Nonce` field as follows: "A 64-bit value that is now deprecated due to the replacement of proof of work consensus. It is set to 0x0000000000000000; formally Hn."

For this reason, usage of a dummy value of all zero might be better.

Environment variables ignored in `NewService`

In the sdk repository, the following function in sdk/prover/server.go

```
func NewService(app sdk.AppCircuit, config ServiceConfig) (*Service, error) {
    pk, vk, ccs, err := readOrSetup(app, config.SetupDir, config.GetSrsDir())
    if err != nil {
        return nil, err
    }
    return &Service{
        svr: newServer(app, pk, vk, ccs),
    }, nil
}
```

should likely use `config.GetSetupDir()` instead of `config.SetupDir` to handle usage of environmental variables in the path correctly, similarly to how it is done for the SRS dir with `config.GetSrsDir()`.

Dead code in Poseidon implementation

The function `MixS` in the zk-hash repository's poseidon/circuit/circuit.go is never used anywhere and could thus be removed.

Likely unnecessary check in `Bytes2Bits`

In the zk-hash repository, the `Bytes2Bits` function in keccak/periphery.go is implemented as follows:

```
func Bytes2Bits(bytes []byte) (bits []uint8) {
    if len(bytes)%8 != 0 {
        panic("invalid length")
    }
    for i := 0; i < len(bytes); i++ {
        bits = append(bits, byte2Bits(bytes[i])...)
    }
    return
}
```

It is unclear why the check that 8 divides `len(bytes)` is performed in this function. If this check is not needed, it should be removed so that the function also becomes usable in other cases. Should this check have meaning in the context of a particular use (for example, when serializing data into a sequence of 64-bit words), consider creating a wrapper with a name reflecting the use case and moving the check into the wrapper.

Keccak functions require round indexes while padding requires input length

The zk-hash repository offers multiple functions to calculate the Keccak hash (`Keccak256BitsOptimized`, `Keccak256Bits`, and `Keccak256`). These require a round index as an argument. This argument `roundIndex` can be a circuit variable and not just a native constant, so the Keccak functions support variable-length inputs in circuit. The in-circuit padding function `Pad101Bits` instead gets the length of the input as an argument. This `inLen` argument can also be a circuit variable, so `Pad101Bits` supports variable length inputs as well. However, there is no in-circuit implementation of the calculation of the round index from the input length that a user could use. There is a `GetRoundIndex` function in keccak/periphery.go, but it only carries out this calculation for the native `int` type. It might make sense to add a function that constrains the calculation of the round index from the input length in circuit.

4.5. Code duplication

We observed several instances of duplicated code within and across repositories. For example, various functions were duplicated across the original scopes in the repos zk-bridge and zk-utils. In the scopes ultimately audited, we observed less duplication; however, some duplication is still present. For example, in the sdk repo, common/utils/binary.go only contains a function that exists identically in the zk-hash repo at utils/binary.go, which could be used instead. In zk-hash, the files keccak/keccakf/keccakf_old.go and keccak/keccakf/keccakf.go are mostly duplicate.

We recommend to avoid code duplication and instead refactor code to use a single implementation to prevent mistakes in the future, should the two implementations diverge.

4.6. Consistent usage of named constants

In some cases, we noticed that named constants were replaced by literal constants with the same value in some places. Using the named constants everywhere would make the code more robust by preventing changes that lead to inconsistencies. In this section, we collect instances of this type.

Maximum number of hash rounds for the fabric's header circuit

In the zk-bridge file circuits/fabric/headers/circuit.go, a constant `CHUNK_HEADER_MAX_HASH_ROUNDS` is used for the maximum number of hash rounds, and this is what is used with regards to what the length of the `Headers` field of the `Circuit` struct should be:

```
const CHUNK_HEADER_MAX_HASH_ROUNDS = 5

type Circuit struct {
```

```
    ChunkRoot [2]frontend.Variable `gnark:",public"`
    PrevHash [2]frontend.Variable `gnark:",public"`
    EndHash [2]frontend.Variable `gnark:",public"`
    StartBlockNum frontend.Variable `gnark:",public"`
    EndBlockNum frontend.Variable `gnark:",public"`

    // RLP encoded header bits, pre-padded with 10..1 for keccak, additional
        zeros are padded in the end
    // to fill up to maxRounds * 1088
    Headers [][]frontend.Variable
    HashRoundIdxs []frontend.Variable

    api frontend.API
}
```

The named constant `CHUNK_HEADER_MAX_HASH_ROUNDS` has value `5`. In circuits/fabric/headers/-headerutil/utils.go, the `EncodeHeaders` function is the one ultimately producing the `Headers` field, but instead of using `CHUNK_HEADER_MAX_HASH_ROUNDS`, it uses a hardcoded constant literal `5`:

```
func EncodeHeaders(headers []types.Header, dummyHeaders bool) (encoded
    [][]frontend.Variable, idxs []frontend.Variable, err error) {
    for i, header := range headers {

        // ...

        zerosToPad := 5*1088 - len(paddedBits)
        for i := 0; i < zerosToPad; i++ {
            fv = append(fv, 0)
        }
        encoded = append(encoded, fv)
    }
    return
}
```

We recommend to use `CHUNK_HEADER_MAX_HASH_ROUNDS` here instead of `5` to ensure consistency on future changes.

4.7. Lists and tuples' ambiguous behavior when used recursively

In sdk/variable.go of the sdk repository, the `List[T CircuitVariable]` type itself implements `CircuitVariable`. However, should lists be nested, in the following implementation of `FromValues`, the loop will never terminate, as fresh lists are empty — so `nv` will be zero.

```
func (l List[T]) FromValues(vs ...frontend.Variable) CircuitVariable {
    typ := *new(T)
    nv := int(typ.NumVars())
    for i := 0; i < len(vs); i += nv {
        values := vs[i : i+nv]
        l[i] = l[i].FromValues(values...).(T)
    }
    return l
}
```

Additionally, the above implementation of `FromValues` will also fail for any other types implementing `CircuitVariable`, unless all the components of the list have the same positive number of field elements and this size is also the same as the size of the default of type `T`.

Similarly, tuples can also not have components that are lists, for the same reason.

We recommend to clearly document these limitations.

4.8. Behavior of `bits2Bytes` and `addOutput` for circuit variables

In the sdk repository, sdk/host_circuit.go includes the following function `bits2Bytes`:

```
func bits2Bytes(data []frontend.Variable) []byte {
    var bits []uint
    for _, b := range data {
        bits = append(bits, uint(fromInterface(b).Int64()))
    }

    bytes := make([]byte, len(bits)/8)
    for i := 0; i < len(bits)/8; i++ {
        for j := 0; j < 8; j++ {
            bytes[i] += byte(bits[i*8+j] << j)
        }
    }

    return bytes
}
```

Note that this function calls the `fromInterface` function of Finding 3.8. ↗ with every component of the argument `data`. Currently, this will return zero on components of `data` that are of an unsupported type, such as circuit variables. For Finding 3.8. ↗, we recommend that `fromInterface` panic on unsupported types instead of returning zero. Should this change be implemented, this will result in `bits2Bytes` panicking if any of the components of `data` is not of a type supported by `fromInter-`

face. Taking into account only bits2Bytes itself, this is reasonable behavior.

However, bits2Bytes is called by the following function from sdk/circuit_api.go:

```
func (api *CircuitAPI) addOutput(bits []variable) {
    // the decomposed v bits are little-endian bits. The way evm uses Keccak
    expects
    // the input to be big-endian bytes, but the bits in each byte are little
    endian
    b := flipByGroups(bits, 8)
    api.output = append(api.output, b...)
    dryRunOutput = append(dryRunOutput, bits2Bytes(b)...)
}
```

This function will be run as part of the application circuit's Define for outputs of that circuit. This will both happen for dry runs with native types as well as with circuit variables during construction of the constraint system in order to, for example, generate the verification key.

In the former case, the relevant effect is that dryRunOutput is updated using bits2Bytes. This works in that case, as the types that ultimately reach fromInterface should be native types that it supports.

However, in the latter case, the types will be circuit variables, which fromInterface does not support. In this case, correct values for dryRunOutput are not needed, so it is fine that so far bits2Bytes will return incorrect values.

Should fromInterface be changed, as recommended in Finding 3.8. ↗, then bits2Bytes and thus addOutput will panic when run with circuit variables. Because of this, the code would need to be structured differently so as to only call bits2Bytes when actually in the case of a dry run — but not otherwise.

4.9. Incorrect hex string and parsing errors canceling each other out

In the sdk repository's sdk/app.go, the function assignInputCommitment has a comment saying that 0 is assigned instead of an actual hash for dummy data:

```
// assign 0 to input commit for dummy and actual data hash for non-dummies
```

However, the hash is assigned in the following line

```
leafs[j] = new(big.Int).SetBytes(common.Hex2Bytes("0x01"))
```

which appears as though the byte value 1 is assigned, not 0. The comment is correct here that 0 should be assigned instead of 1, as can be seen by comparison to the commitInput function in

host_circuit.go.

Even though it does not appear as so, the above line does assign the value `0`. The reason is that `common.Hex2Bytes` ultimately passes the argument through to Go's `Decode` function in the standard library's encoding/hex/hex.go. This function only accepts hexadecimal digits as characters, so `0` through `9`, `a` through `f`, and `A` through `F`. There is no support for `0x` prefixes, so an error will be returned when attempting to parse the character `x`:

```
// Decode decodes src into [DecodedLen](len(src)) bytes,
// returning the actual number of bytes written to dst.
//
// Decode expects that src contains only hexadecimal
// characters and that src has even length.
// If the input is malformed, Decode returns the number
// of bytes decoded before the error.
func Decode(dst, src []byte) (int, error) {
    i, j := 0, 1
    for ; j < len(src); j += 2 {
        p := src[j-1]
        q := src[j]

        a := reverseHexTable[p]
        b := reverseHexTable[q]
        if a > 0x0f {
            return i, InvalidByteError(p)
        }
        if b > 0x0f {
            return i, InvalidByteError(q)
        }
        dst[i] = (a << 4) | b
        i++
    }
    // ...
}
```

This `Decode` function is called through `DecodeString`, which is a thin wrapper passing the error upwards. Instead of returning the byte index at which the parsing error occurred, like `Decode`, this wrapper returns the bytes decoded so far:

```
// DecodeString returns the bytes represented by the hexadecimal string s.
//
// DecodeString expects that src contains only hexadecimal
// characters and that src has even length.
// If the input is malformed, DecodeString returns
// the bytes decoded before the error.
func DecodeString(s string) ([]byte, error) {
```

```
    src := []byte(s)
    // We can use the source slice itself as the destination
    // because the decode loop increments by one and then the 'seen' byte is
    not used anymore.
    n, err := Decode(src, src)
    return src[:n], err
}
```

Finally, the `common.Hex2Bytes` that is called with `0x01` calls `DecodeString`, but the error is ignored:

```
// Hex2Bytes returns the bytes represented by the hexadecimal string str.
func Hex2Bytes(str string) []byte {
    h, _ := hex.DecodeString(str)
    return h
}
```

Thus, with `0x01`, as there will be a parsing error at the `x`, where no byte has been fully parsed yet, the value of `h` will be the empty slice. When calling `new(big.Int).SetBytes` with an empty slice, the integer value obtained from it will be zero.

This explains why

```
leafs[j] = new(big.Int).SetBytes(common.Hex2Bytes("0x01"))
```

will set `leafs[i]` to zero, as it should, even though the code looks like it does not.

Note that in the sdk repository's common/utils/hex.go, there is also a function called `Hex2Bytes`, which could be confused with the function `common.Hex2Bytes` discussed so far, which comes from the go-ethereum package. Brevis's `Hex2Bytes` is implemented as follows:

```
// Hex2Bytes supports hex string with or without 0x prefix
// Calls hex.DecodeString directly and ignore err
// similar to ec.FromHex but better
func Hex2Bytes(s string) (b []byte) {
    if len(s) >= 2 && s[0] == '0' && (s[1] == 'x' || s[1] == 'X') {
        s = s[2:]
    }
    // hex.DecodeString expects an even-length string
    if len(s)%2 == 1 {
        s = "0" + s
    }
    b, _ = hex.DecodeString(s)
    return b
}
```

As seen, this function *does* support the `0x` prefix. Should `common.Hex2Bytes` in

```
leafs[j] = new(big.Int).SetBytes(common.Hex2Bytes("0x01"))
```

be replaced by a call to this function `Hex2Bytes`, then `leafs[i]` would be assigned `1`, which would be incorrect.

Currently, usage of go-ethereum's `Hex2Bytes` — without ensuring there are no parsing errors — and the incorrect `01` instead of `00` essentially amount to two mistakes that happen to cancel each other out and lead to the correct result. This code can be very confusing to future developers though, which may inadvertently introduce mistakes later. Presence of the local function with the same name that behaves differently makes this particularly risky.

We thus recommend to replace `common.Hex2Bytes("0x01")` by either `common.Hex2Bytes("00")` or a call to the common/utils/hex.go version of `Hex2Bytes` with argument `"0x00"`.

4.10. References to MiMC instead of Poseidon

In several places in the codebase, the MiMC hash function is named when in fact the Poseidon hash function is used. This can at times make the code confusing to follow and misleading, and thus it could cause mistakes. We list these occurrences below.

Merkle tree root-hash calculation in `assignInputCommitment` and `assignToggleCommitment`

In the sdk repository's sdk/app.go, the `assignInputCommitment` and `assignToggleCommitment` functions use the Poseidon hash for calculating the commitments and the Merkle tree root hash. However, some variable names are referring to MiMC instead. Additionally, `mimc_bn254.BlockSize`, a constant genuinely arising from a MiMC implementation, is used. This constant happens to have a value that is compatible with its use, so the code currently functions correctly, nevertheless. The following snippet from `assignToggleCommitment` shows the confusing parts of the two functions (`assignInputCommitment` is analogous):

```
func (q *BrevisApp) assignToggleCommitment(in *CircuitInput) {
    // ...

    hasher := utils.NewPoseidonBn254()

    // ...

    for {
        if elementCount == 1 {
```

```
            in.TogglesCommitment = leafs[0]
            return
        }
        for i := 0; i < elementCount/2; i++ {
            var mimcBlockBuf0, mimcBlockBuf1 [mimc_bn254.BlockSize]byte
            leafs[2*i].FillBytes(mimcBlockBuf0[:])
            leafs[2*i+1].FillBytes(mimcBlockBuf1[:])
            hasher.Reset()
            hasher.Write(new(big.Int).SetBytes(mimcBlockBuf0[:]))
            hasher.Write(new(big.Int).SetBytes(mimcBlockBuf1[:]))
            result, err := hasher.Sum()
            if err != nil {
                panic(fmt.Sprintf("failed to hash merkle tree: %s",
    err.Error()))
            }
            leafs[i] = result
        }
        elementCount = elementCount / 2
    }
}
```

We recommend to rename the variables and replace `mimc_bn254.BlockSize` by a suitable constant relating to Poseidon (for example by adding a `BlockSize` constant to utils/poseidon_bn254.go in the zk-hash repository).

As a side remark, code duplication could be reduced by factoring out the root-hash computation that is currently present in both `assignInputCommitment` and `assignToggleCommitment` into a separate function.

Comment in `BuildCircuitInput`

Also in sdk/app.go, the `BrevisApp.BuildCircuitInput` starts with the following comments:

```
func (q *BrevisApp) BuildCircuitInput(app AppCircuit) (CircuitInput, error) {

    // 1. mimc hash data at each position to generate and assign input
    commitments and toggles commitment
    // 2. dry-run user circuit to generate output and output commitment
```

The first comment line should use "Poseidon hash data" instead of "mimc hash data".

Documentation comments for functions in poseidon_bn254_circuit.go

In the zk-hash repository's file poseidon/poseidon_bn254_circuit.go, the type `PoseidonCircuit` and function `NewBn254PoseidonCircuit` have comments using "MiMC" where they should use "Poseidon":

```
// MiMC contains the params of the Mimc hash func and the curves on which
    it is implemented
type PoseidonCircuit struct {
    preimage []frontend.Variable // state storage. data is updated when
    Write() is called. Sum sums the data.
    api frontend.API // underlying constraint system
}

// NewMiMC returns a MiMC instance, that can be used in a gnark circuit
func NewBn254PoseidonCircuit(api frontend.API) (PoseidonCircuit, error) {
    return PoseidonCircuit{
        api: api,
    }, nil
}
```

4.11. Unallocated inputs slots not constrained

The host circuit holds a slice of circuit variables `Input.InputCommitments` for commitments to input data (receipt, storage, or transaction data). This is instantiated in with the `defaultCircuitInput` function of sdk/circuit_input.go with a fixed length `NumMaxDataPoints` and default value `0` for all entries. A number of entries corresponding to the allocated number of receipts, storage, and transactions is assigned to with `BrevisApp.assignInputCommitment` in sdk/app.go.

These entries corresponding to allocated receipts, storage, and transactions are constrained to be the commitments of the corresponding input data used in the circuit. The entire list of input commitments is used as leaves of a Merkle tree whose root hash is exposed as a public input to the circuit.

Except their involvement in the Merkle root-hash calculation, the `Input.InputCommitments` entries beyond the allocated ones are not constrained, however. This does not appear to be a problem in the current circuits.

The lack of constraints means that it is possible to assign arbitrary commitment hashes there. Like this, it may be possible to add additional input data entries beyond the maximum the circuit declared. These entries still need to be legitimate, however, because they take part in the root hash that is ultimately checked by the Brevis backend circuits.

Additionally, application circuits using the sdk as intended should not access these extra data commitments, and there will be no corresponding unpacked data for these entries in the circuit.

For defense in depth, we recommend to nevertheless consider constraining these unused data commitments to be zero, which would take only a small number of constraints.

Alternatively, it may be possible to use constants for these entries instead of circuit variables, in which case no additional constraints would be needed, and witnesses could be saved as well. The way to do this should be to change the `defaultCircuitInput` function in sdk/circuit_input.go to instantiate the `inputCommits` slice with the total allocated length rather than the number of leaves for the eventual Merkle tree:

```
func defaultCircuitInput(maxReceipts, maxStorage, maxTxs int) CircuitInput {
   numTotalAllocated := maxReceipts + maxStorage + maxTxs
      var inputCommits = make([]frontend.Variable, NumMaxDataPoints)
      var inputCommits = make([]frontend.Variable, numTotalAllocated)
      for i := 0; i < NumMaxDataPoints; i++ {
      for i := 0; i < numTotalAllocated; i++ {
          inputCommits[i] = 0
      }
      return CircuitInput{
          DataInput: defaultDataInput(maxReceipts, maxStorage, maxTxs),
          InputCommitmentsRoot: 0,
          InputCommitments: inputCommits,
          TogglesCommitment: 0,
          OutputCommitment: OutputCommitment{0, 0},
      }
}
```

This will require additional corresponding changes in other parts of the code. In particular, in `HostCircuit.Define`, the line

```
inputCommitmentRoot, err := calMerkelRoot(gapi, c.Input.InputCommitments)
```

would need to be modified. The second argument to `calMerkelRoot` would now need to consist not of `c.Input.InputCommitments` but of `c.Input.InputCommitments` padded with the constant 0 up to length `NumMaxDataPoints`. There may also be other spots in the codebase requiring modifications.

4.12. Keccak padding function `pad10*1`

For the keccak hash function, the input bit string needs to be padded to a length that is a multiple of 1,088. This padding function, called `pad10*1`, is specified in section 5.1 of the SHA-3 Standard ↗. It can be described as follows: a bit 1 is appended, and another (additional) bit 1 will be added at the very end, and then enough 0s (possibly none) are inserted between these two 1 bits to make the length

divisible by 1,088. If `bits` is the list of `bits` that need to be padded, then the number of zeroes is thus `(-len(bits)-2) modulo 1088`, or in Go (due to Go's behavior with respect to remainders of negative numbers), `(1088 - ((len(bits) + 2) % 1088)) % 1088`. The number of 1,088-bit blocks of the padded data are then `ceil((len(bits) + 2) / 1088) = floor((len(bits) + 1) / 1088) + 1`.

4.13. Confusing function `GroupBy`

The sdk offers the following function in sdk/datastream.go:

```
// GroupBy a given field (identified through the field func), call reducer on
// each group, and returns a data stream in which each element is an aggregation
// result of the group. The optional param maxUniqueGroupValuesOptional can be
// supplied to optimize performance. It assumes the worst case (all values in
    the
// data stream are unique) if no maxUniqueGroupValuesOptional is configured.
func GroupBy[T, R CircuitVariable](
    ds *DataStream[T],
    reducer ReduceFunc[T, R],
    reducerInit R,
    field GetValueFunc[T],
    maxUniqueGroupValuesOptional ...int,
) (*DataStream[R], error) {
    // ...
}
```

It is unclear based only on the documentation comment what exactly this function is intended to do, in particular as it is not explained what is meant by a group.

From reading the implementation, it appears that the groups mentioned in the documentation refer to the elements of the data stream with the same value.

Let us go through the implementation.

```
if len(maxUniqueGroupValuesOptional) > 1 {
    panic("invalid amount of optional params")
}
g := ds.api.g
values := make([]frontend.Variable, len(ds.underlying))
for i, v := range ds.underlying {
    values[i] = field(v).Val
}
groupValues, err := computeGroupValuesHint(g, values, ds.toggles)
if err != nil {
```

```
    return nil, err
}
```

So far, `values` stores the underlying data from the data stream after applying `field`, and the components of `groupValues` are witnesses that are so far not constrained and can be set to anything by the prover. The hint assigning these values is implemented so as to return a list of unique components of `values`, padded with zero. Note, however, that a malicious prover can assign something else as well.

The data stream that will eventually be returned is made up of data `aggResults` and toggles `aggResultToggles`, of length `maxGroupValues`, which is the full length of the values if the user did not supply a length with the optional argument `maxUniqueGroupValuesOptional`:

```
maxGroupValues := len(values)
if len(maxUniqueGroupValuesOptional) == 1 {
    maxGroupValues = maxUniqueGroupValuesOptional[0]
}
aggResults := make([]R, maxGroupValues)
aggResultToggles := make([]frontend.Variable, maxGroupValues)
```

The following comment then explains why it is accepted that a malicious prover can assign different `groupValues` than intended:

```
// Filter on each groupValue, then reduce using the user supplied function.
    Note
// that the groupValues are computed using hints. This is ok because the
// groupValues are used as predicates in the filter step. Assuming the
// outside-of-circuit-computed groupValues are all malicious (does not exist in
// the input data stream), then the filter step would result in empty data
// streams, and the reduce results would be all toggled off. This decision is
// made based on the premise of which it is accepted that we can't prove what's
// NOT provided to the circuit as inputs.
```

We then arrive at the actual logic to construct the results' data stream.

```
for i := 0; i < maxGroupValues; i++ {
    vs := groupValues[i]
    group := Filter(ds, func(current T) Uint248 {
        v := field(current)
        return newU248(g.IsZero(g.Sub(v.Val, vs)))
    })
```

The original data stream is filtered to only have those entries enabled that match the current `groupValues` component after applying `field`.

Then the reduction is applied to this filtered data stream:

```
aggResults[i] = Reduce(group, reducerInit, reducer)
```

Finally, the toggle for the result is computed as described in the comment:

```
    // only turn on toggles for agg result if the filtered group has at least 1
    item
    aggResultToggles[i] = g.Sub(1, g.IsZero(Count(group).Val))
}
return newDataStream(ds.api, aggResults, aggResultToggles), nil
```

Thus, one could describe the function as follows.

The prover can supply a list of up to `maxUniqueGroupValuesOptional[0]` (or `len(ds.underlying)` if `maxUniqueGroupValuesOptional` is not passed) arbitrary values. The resulting data stream will then at component `i` have the result of the reduction computed over as many copies of the `i`th value supplied by the prover as there are in `ds`. If there are no copies in `ds`, then the component is toggled off.

Note that in particular, the prover can always choose values that do not occur, and thereby produce an empty (as in everything toggled off) result. Also, the prover can use values multiple times, and the aggregations will then also occur multiple times in the result.

Outside of a test, there is no usage example for `GroupBy` in the codebase. It is unclear how `GroupBy` should be used given that the caller does not get any information on which values `ds` was filtered for, only the aggregations. It seems like it might be more useful to, for example, make `aggResults[i]` a tuple containing both `Reduce(group, reducerInit, reducer)` and `vs`, so that the caller knows which value was filtered and reduced over. Perhaps it might also make sense to ensure that the values in `groupValues` chosen by the prover are all unique. This might run into issues with the value zero used for padding, however, as it would be unclear how to distinguish between zero used as padding and zero used as one of the values.

We recommend to clarify intended functionality and usage of `GroupBy` with additional documentation and possibly an example.

4.14. Setup

The sdk provides several helper functions for performing the setup for the circuits. In this section, we collect some observations about this.

Document that `Setup` should only be used for testing

The file sdk/periphery.go contains a function `Setup` that generates and saves an SRS that is suitable for the constraint system, using the package github.com/consensys/gnark/test/unsafekzg. We recommend to document that this `Setup` function (and functions that use it) are only safe for using for tests and that for production use, users should use an SRS generated by, for example, an MPC ceremony (possibly downloading such an SRS with functionality in sdk/srs/srs.go).

Dowloaded SRS not checked to be legitimate

The `download` function in sdk/srs/srs.go can be used (via `NewSRS`) to download an SRS. The downloaded file is used as is, with no validation done or check of authenticity. To protect against supply-chain attacks, we recommend to distribute a list of known hashes of SRS files with the sdk and then check downloads against this list. For users that wish to use other SRSs, documentation might be added explaining to users how to manually place their SRS into the cache directory.

No validation

SRSs are not validated to be well-formed but read with `UnsafeReadFrom`, which does not check that the points read are in the correct subgroup. As the data needs to be trusted anyway, given that even well-formed SRSs might be insecure if someone knows the toxic waste, this may be a reasonable trade-off, if the origin of the SRS is trusted.

No recovery on incompletely written files

SRSs are cached on disk, with the filename derived from the SRS size. Should writing an SRS file to disk be interrupted, there will be an error when attempting to read this cached SRS. This error will be caught, and instead, an SRS will be downloaded. This new SRS will, however, not be written to disk (overwriting the broken SRS), as the file already exists. Thus to be able to cache an SRS of this size again, it will be necessary to manually remove the broken file. We recommend to document this and add a printout to `ReadFile` in case the file is found but reading an SRS from it fails, in order to alert users to this case, as currently visible behavior does not distinguish between the file not having been present and having been present but broken in some way.

4.15. Verification of Poseidon constants

The zk-hash repository contains an in-circuit Poseidon implementation in poseidon/circuit/circuit.go, with associated constants in poseidon/circuit/constants.go. The implementation does not follow the definition of Poseidon from the main text of the Poseidon paper ↗ directly. Instead, an optimization is used, where in particular mixing between non–S-box parts of the state in the partial rounds is moved outside of the partial rounds, and some other changes to the algorithm are done. This is done to improve efficiency, as this reduces the number of operations that need to be done

in circuit. Essentially, the formulas are rearranged so as to obtain terms that involve only constants, so that these terms can be precomputed. Such optimizations are discussed in the paper as well, in Appendix B.

The Poseidon paper's authors provide scripts ↗ that can be used to calculate constants to be used in Poseidon implementations. These scripts provide constants that fit with the standard description used in the main text of the paper, not the different constants needed for the particular optimized algorithm used by the implementation in poseidon/circuit/circuit.go.

To verify that the implementation in zk-hash is equivalent to the paper authors' specification and constants, we wrote a script that calculates the coefficients of the linear dependency of each part of the state on certain previous words of states. Doing this for both the standard algorithm and constants, and Brevis's algorithm and constants, we can check that they are equal step by step, thereby at the end confirming that the two algorithms compute the same function.

To run our scripts, it is first necessary to clone the paper authors' repository at https://extgit.iaik.tugraz.at/krypto/hadeshash/-/tree/master ↗. All further description will be local in the `code` directory of that repository.

A generate_params_poseidon.sage script is provided to generate parameters for Poseidon instances. However, Brevis uses a different number of partial rounds by rounding the suggested number up to the nearest multiple of `t`, the number of field elements the Poseidon permutation is defined for. Thus, the script can be used to obtain the partial round numbers, to check that the ones Brevis uses are correctly rounded up, but the script needs to be patched to produce constants needed for Brevis instantiation of Poseidon. The following patch makes the necessary changes:

```
-- generate_params_poseidon.sage 2024-10-14 21:00:10.632136143 +0200
++ generate_params_poseidon_fixed_r_p.sage 2024-10-17 23:21:55.858042766
    +0200
@@ -2,8 +2,8 @@
 import sys
 from sage.rings.polynomial.polynomial_gf2x import GF2X_BuildIrred_list

if len(sys.argv) < 8:
    print("Usage: <script> <field> <s_box> <field_size> <num_cells> <alpha> <
        security_level> <modulus_hex>")
if len(sys.argv) < 9:
    print("Usage: <script> <field> <s_box> <field_size> <num_cells> <alpha> <
        security_level> <modulus_hex> <R_P>")
     print("field = 1 for GF(p)")
     print("s_box = 0 for x^alpha, s_box = 1 for x^(-1)")
     exit()
@@ -47,6 +47,8 @@
 else:
     print("Unknown field type, only 0 and 1 supported!")
     exit()
```

```
R_P_REQUESTED = int(sys.argv[8])


 def sat_inequiv_alpha(p, t, R_F, R_P, alpha, M):
     N = int(FIELD_SIZE * NUM_CELLS)
@@ -144,6 +146,9 @@
 ROUND_NUMBERS = calc_final_numbers_fixed(PRIME_NUMBER, NUM_CELLS, ALPHA,
    SECURITY_LEVEL, True)
 R_F_FIXED = ROUND_NUMBERS[0]
 R_P_FIXED = ROUND_NUMBERS[1]
assert R_P_FIXED <= R_P_REQUESTED
R_P_FIXED = R_P_REQUESTED
assert R_F_FIXED == 8
 # R_F_FIXED = 8
 # R_P_FIXED = 60
```

We called the so-changed script "generate_params_poseidon_fixed_r_p.sage".

Given the above changes, the following Python script, placed in the same directory, will use generate_params_poseidon_fixed_r_p.sage to generate the constants needed for the standard Poseidon algorithm. It will then read the file brevis.go to obtain the modified constants used by Brevis. This file, brevis.go, should be the file poseidon/circuit/constants.go from the zk-hash repo, copied into the directory. Finally, the script will carry out checks to ensure that the two Poseidon implementations are equivalent.

```
#!/usr/bin/env python3
import subprocess
from sage.all import matrix, vector, GF, block_matrix, identity_matrix,
    zero_matrix

prime = 0x30644e72e131a029b85045b68181585d2833e84879b9709143e1f593f0000001
F = GF(prime)

def generate_params():
    rounds = [
            None, None,
            56, 57, 56, 60, 60, 63, 64, 63, 60, 66, 60, 65, 70, 60, 64, 68
            ]
    for t in range(2,18):
        subprocess.run(
                ['sage', 'generate_params_poseidon_fixed_r_p.sage',
                 '1', '0', '254', str(t), '5', '128',

    '0x30644e72e131a029b85045b68181585d2833e84879b9709143e1f593f0000001',
                 str(rounds[t])])
```

```
def get_paper_params():
    round_contants = dict()
    MDS_matrices = dict()
    R_Ps = dict()

    for t in range(2,18):
        with open(f'poseidon_params_n254_t{t}_alpha5_M128.txt', 'r') as fh:
            data = fh.read()
        data = data.split('\n')
        R_F = int(data[0].split('R_F=')[1].split(', ')[0])
        R_P = int(data[0].split('R_P=')[1])
        assert R_F == 8
        c = eval(data[5])
        m = eval(data[17])
        for i in range(len(c)):
            c[i] = eval(c[i])
        for i in range(len(m)):
            for j in range(len(m[i])):
                m[i][j] = eval(m[i][j])

        R_Ps[t] = R_P
        round_contants[t] = c
        MDS_matrices[t] = m
    return round_contants, MDS_matrices, R_Ps

def extract_brevis_param_letter(data, letter):
    result = dict()
    data = data.split('func strPOSEIDON_' + letter)[1].split('func ')[0]
    data = data.split('if t == ')[1:]
    for entry in data:
        t = int(entry.split(' ')[0])
        string = entry.split('`')[1]
        if t == 2 and letter == 'C':
            string = string[:-1]
        value = eval(string)
        result[t] = value
    return result

def get_brevis_params():
    with open('brevis.go', 'r') as fh:
        data = fh.read()
    c = extract_brevis_param_letter(data, 'C')
    s = extract_brevis_param_letter(data, 'S')
    m = extract_brevis_param_letter(data, 'M')
    p = extract_brevis_param_letter(data, 'P')
```

```
    return c,s,m,p

def check_for_t(t, c, s, m, p, paper_c, paper_m, r_P):
    m = matrix(F, m)
    p = matrix(F, p)
    paper_m = matrix(F, paper_m)

    # Correctness of first four full rounds except the last Mix need the
    # following.
    assert m.transpose() == paper_m
    assert c[:t] == paper_c[:t]
    for i in range(1,4):
        assert vector(F, c[t*i:t*(i+1)]) == vector(F, paper_c[t*i:t*(i+1)])
    * m.inverse()

    # For the partial rounds, we symbolically check this.
    # We do this from just before the last Mix from the last full round
    # to after the round key addition of the first full round after the
    # partial ones.
    # We use the following variables:
    # - dummy variable for constant 1 (so we can handle that this is
    # not linear, but affine linear)
    # - initial state[i] for i>0
    # - initial state[0], and state[0] at each step just after sigma,
    # 1+r_P in total.
    # and compute the matrices for how each state[0] before a sigma
    # application depends on the previous, and how the resulting state
    # depends on all of the above.
    # This is all linear, so this means just computing some matrices.
    # We then compare this to matrices from the constants.

    # We do the paper first, then Brevis' implementation

    # Initial state. This means before applying the last Mix from the
    # last full round.
    num_vars = 1 + (t - 1) + (1 + r_P)
    state_0 = matrix(F, [0] + [0]*(t-1) + [1] + [0]*r_P).transpose()
    state_positive = block_matrix(F,
                                  [[zero_matrix(F, 1, t - 1)],
                                   [identity_matrix(F, t - 1)],
                                   [zero_matrix(F, 1 + r_P, t - 1)]])
    state = block_matrix(F, [[state_0, state_positive]])

    paper_pre_sigma_state_0 = []

    # First, the last Mix.
    state = state * m
```

```
    for i in range(r_P):
        # Addition of round keys:
        round_keys_this_round = paper_c[(4+i)*t:(4+i+1)*t]
        round_keys_matrix = matrix(
                [round_keys_this_round] +
                [[F(0) for __ in range(t)] for _ in range(num_vars - 1)])
        state = state + round_keys_matrix

        # Now we extract the pre-sigma state_0.
        paper_pre_sigma_state_0.append(state.column(0))

        # We will check that the pre-sigma state 0 is correct separately.
        # We thus continue with the post-sigma state 0 replaced by a variable.
        new_state_0 = matrix(F,
                             [[0]*(1 + t - 1 + 1 + i) +
                              [1] + [0]*(r_P - 1 - i) ]).transpose()
        rest_state = state.submatrix(0, 1, num_vars, t-1)
        state = block_matrix(F, [[new_state_0, rest_state]])

        # Finally, we apply the Mix step.
        state = state * m

    round_keys_this_round = paper_c[(4+r_P)*t:(4+r_P+1)*t]
    round_keys_matrix = matrix(
            [round_keys_this_round] +
            [[F(0) for __ in range(t)] for _ in range(num_vars - 1)])
    state = state + round_keys_matrix

    paper_final_state = state


    # Now we do Brevis' implementation
    state_0 = matrix(F, [0] + [0]*(t-1) + [1] + [0]*r_P).transpose()
    state_positive = block_matrix(F,
                                  [[zero_matrix(F, 1, t - 1)],
                                   [identity_matrix(F, t - 1)],
                                   [zero_matrix(F, 1 + r_P, t - 1)]])
    state = block_matrix(F, [[state_0, state_positive]])

    brevis_pre_sigma_state_0 = []

    # state = Ark(api, state, c, nRoundsF/2*t)
    round_keys_pre_loop = c[(4)*t:(4+1)*t]
    round_keys_matrix = matrix([round_keys_pre_loop] +
                               [[F(0) for __ in range(t)]
                                for _ in range(num_vars - 1)])
    state = state + round_keys_matrix
```

```
    # state = Mix(api, state, p)
    state = state * p

    for r in range(r_P):
        # Now we extract the pre-sigma state_0.
        brevis_pre_sigma_state_0.append(state.column(0))

        # We continue with the post-sigma state 0 replaced by a variable.
        new_state_0 = matrix(F, [[0]*(1 + t - 1 + 1 + r) +
                                 [1] + [0]*(r_P - 1 - r) ]).transpose()
        rest_state = state.submatrix(0, 1, num_vars, t-1)
        state = block_matrix(F, [[new_state_0, rest_state]])

        # state[0] = api.Add(state[0], c[(nRoundsF/2+1)*t+r])
        constants_to_add = [c[(4+1)*t + r]] + [0]*(t-1)
        constants_to_add_matrix = matrix([constants_to_add] +
                                         [[F(0) for __ in range(t)]
                                          for _ in range(num_vars - 1)])
        state = state + constants_to_add_matrix

        # newState0 := frontend.Variable(0)
        # for j := 0; j < len(state); j++ {
        # mul := api.Mul(s[(t*2-1)*r+j], state[j])
        # newState0 = api.Add(newState0, mul)
        # }
        new_states = []
        new_state_0 = vector(F, [0]*num_vars)
        for j in range(t):
            new_state_0 = new_state_0 + F(s[(t*2 - 1)*r + j])*state.column(j)
        new_states.append(new_state_0)

        # for k := 1; k < t; k++ {
        # state[k] = api.Add(state[k], api.Mul(state[0], s[(t*2-1)*r+t+k-1]))
        # }
        for k in range(1, t):
            new_state_k = (state.column(k) +
                           F(s[(t*2 - 1)*r + t + k - 1])*state.column(0))
            new_states.append(new_state_k)

        # state[0] = newState0
        # and also update all the other ones, as we haven't done that yet
        state = matrix(F, new_states).transpose()


    brevis_final_state = state
```

```
    assert len(paper_pre_sigma_state_0) == len(brevis_pre_sigma_state_0)
    assert len(paper_pre_sigma_state_0) == r_P
    for i in range(r_P):
        # Splitting the assert into lines is just to prevent line breaks
        # in the report.
        assert_check = (paper_pre_sigma_state_0[i] ==
                        brevis_pre_sigma_state_0[i])
        assert_msg = f'pre-sigma state does not match for {i=}'
        assert assert_check, assert_msg
    assert paper_final_state == brevis_final_state

    for i in range(1,4):
        r = i - 1
        # state = Ark(api, state, c, (nRoundsF/2+1)*t+nRoundsP+r*t)
        assert_lhs = vector(F, c[t*(5 + r) + r_P:t*(5 + r + 1) + r_P])
        assert_rhs = (vector(F, paper_c[t*(4+r_P+i):t*(4+r_P+i+1)]) *
                      m.inverse())
        assert assert_lhs == assert_rhs

    print(f'All correct for {t=}')


def check():
    c,s,m,p = get_brevis_params()
    paper_c, paper_m, paper_r_p = get_paper_params()
    rounds = [None, None,
              56, 57, 56, 60, 60, 63, 64, 63, 60, 66, 60, 65, 70, 60, 64, 68]
    for t in range(2, len(rounds)):
        #assert rounds[t] >= paper_r_p[t]
        #assert rounds[t] < paper_r_p[t] + t
        #assert rounds[t] % t == 0
        assert rounds[t] == paper_r_p[t]

        check_for_t(
                t, c[t], s[t], m[t], p[t],
                paper_c[t], paper_m[t], rounds[t])

generate_params()
check()
```

4.16. Test suite

For a large codebase with multiple moving parts and dependencies, comprehensive testing is essential.

Therefore, we recommend building a rigorous test suite that tests all functionalities, not just with integration tests but also through tests of individual functions, to ensure that the system operates as intended.

During our audit, we noticed that while tests were present for many functions, they often only tested particular hardcoded inputs. We recommend to, as far as possible, test random samples across the entire range of possible inputs, along with possibly hardcoded, handpicked test inputs for rare edge cases. For example, Finding 3.10. ↗ could have been found by tests if the Keccak padding function had been tested with inputs of varying lengths, by comparing against external implementations of Keccak and its padding function.

Note that for proof circuits, tests can generally only test for completeness. The most serious bugs are often underconstraints that make the circuit unsound, and these bugs cannot be found with normal testing. Thus, comprehensive testing should be considered complementary to thorough manual review.

5. Assessment Results

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

During our assessment on the scoped Brevis circuits, we discovered 35 findings. Four critical issues were found. Three were of high impact, three were of medium impact, 15 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.