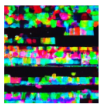


ZKSECURITY

Audit of Linea's gnark std

May 20th, 2024

Introduction

On May 20th, 2024, Linea requested an audit of the gnark std library. The audit was conducted by three zkSecurity engineers over a period of three weeks. The audit focused on the correctness and security of parts of the gnark std library.

We found the documentation and the code to be of high quality, and the team was especially helpful in discussing the various issues brought up during the engagement.

Scope

The scope of the audit included the following components:

- **Non-native arithmetic for fields**, which is used to emulate various foreign fields, their elements and their operations in a circuit.
- **Non-native arithmetic for curves**, which is used to implement elliptic curves.
- **Multiplexers**, which is used to dynamically index or range over a fixed-sized array of values in a circuit.
- **Range checks API**, which is used to check if a value is within a given range in a circuit.
- **In-circuit Plonk verifier**, which is used to verify a Plonk proof in a circuit.
- **In-circuit KZG verifier**, which is used to batch verify KZG polynomial commitments and evaluation proofs by the in-circuit Plonk verifier.

Before we list the findings, we give a brief overview of some of the components of the gnark std library that were audited.

Non-native Arithmetic Implementation

Simulating operations of non-native fields in the native field (i.e. the circuit field) can be problematic as the values might be larger than the circuit field, and the operations might also lead to wrap-arounds which would lead to an incorrect result.

The general idea behind implementations of non-native arithmetic operations is to represent the non-native field elements as a number of limbs in the native field. For example, an element in the base field (or scalar field) of **secp256k1** (~256 bits) can fit into 4 limbs of 64-bit integers, which can be represented as field elements in our circuit.

Then, operations are performed on the limbs themselves, potentially overflowing them, and the result is then reduced modulo the prime of the native field.

Reduction

The idea of doing a modular reduction from a value a (that might have become bigger than the non-native modulus f) to a value b, is to prove the following:

$$a = b \mod f$$

This is essentially the same as proving that we have the values b and k in the integers, such that $b, k < f$:

$$a = b + k \cdot f$$

Now, if we replace the variables with their limbs (of t bits) we obtain something like this:

$$\sum_i a_i \cdot 2^{ti} = \sum_i b_i \cdot 2^{ti} + (\sum_i k_i \cdot 2^{ti}) \cdot (\sum f_i \cdot 2^{ti})$$

The first idea is that the limbs b_i and k_i are provided as a hint by the prover, are range-checked (using table lookups), and then the equation is checked.

But checking the equation as is is not possible as we already know that the modulus f is too large for our circuit field.

So the second idea is to represent the different values as polynomials with their limbs as coefficients. (This way we can instead check that the left-hand side polynomial is equal to the right-hand side polynomial.) For example, for a we have the polynomial:

$$a(x) = \sum_i a_i \cdot x^i$$

Note that this polynomial only takes the value a for $x = 2^t$.

So checking our previous equation, is reduced to checking that the following equation is true at the point $x = 2^t$:

$$a(x) = b(x) + k(x) \cdot f(x)$$

Since 2^t is a root, we know that there exist a polynomial $c(x)$ such that:

$$a(x) - b(x) + k(x) \cdot f(x) = c(x) \cdot (x - 2^t)$$

So the third idea is to let the prover hint this polynomial $c(x)$, and to let them show us that this identity exists. We can either do that at $d + 1$ points if the maximum degree of the polynomials is d, or we can do that at a single random point using the **Schwartz-Zippel lemma**.

The latter solution is what gnark does, using an externalized Fiat-Shamir protocol (which we go over in **Out-of-circuit Fiat-Shamir with Plonk**) to compute a random challenge based on all of the input to the identity check.

Furthermore, since the same challenge is used for ALL field operations in a circuit, the computation of the challenge AND the identity checks are deferred to the end of the circuit compilation, once the externalized Fiat-Shamir protocol has amassed all of the inputs.

Multiplication

Multiplication is pretty straight forward, using the previous technique we check that $a \cdot b = c \mod f$ by checking the following identity:

$$a(x) \cdot b(x) = r(x) + k(x) \cdot f(x) + c(x) \cdot (x - 2^t)$$

Thus, after evaluating the different polynomials at some random point α, we check that

$$a(\alpha) \cdot b(\alpha) = r(\alpha) + k(\alpha) \cdot f(\alpha) + c(\alpha) \cdot (\alpha - 2^t)$$

Note that $f(\alpha)$ and $(\alpha - 2^t)$ only need to be computed once for all multiplication checks.

Subtraction bounds

We are trying to perform $a - b \mod p$ on limbs of t bits.

To avoid underflow, we want to increase every limb a_i with some padding u_i such that:

1. they won't underflow in the integers: $(a_i + u_i) - b_i > 0$.
2. the overall padding $u = \sum_i u_i \cdot 2^{ti}$ won't matter in the equation as it'll be a multiple of the modulus: $u = k \cdot p$

To satisfy 2, they create a value $u_1 > p$, then find its inverse of modulo p by computing as `u_2 = (-u_1) % p`. Since we have that $u_1 > p$ and $u_2 < p$, we also have that $u_1 + u_2 = k \cdot p$ for some $k > 0$.

To satisfy 1, u_1 and u_2 are created such that $u_i = u_1 + u_2 > b_i$.

First, u_1 is created by setting its limbs to an overflowed value: $u_{1,i} = 2^t$

Second, u_2 is created as its negated congruent value as explained above: `u_2 = (-u_1) % p`, and decomposed into t-bit limbs such that $u_{2,i} < 2^t$ for all i.

1. we need to prove that $u_i > b_i$
2. which is equivalent to proving that $u_{1,i} + u_{2,i} > b_i$
3. by construction, $2^t > b_i$
4. $u_{1,i} + u_{2,i} \geq 2^t$, since $u_{1,i} = 2^t$ and $u_{2,i} < 2^t$
5. by 3 and 4, we have that $u_{1,i} + u_{2,i} > b_i$

One can easily extend the proof to include b's potential overflow o, so that we have $2^{t+o} > b_i$ and set $u_{1,i} = 2^{t+o}$.

Overflows

When adding two limbs of n-bit together, the maximum value obtained can be of $n + 1$ bits. For the simple reason that using the maximum values we see that:

$$(2^n - 1) + (2^n - 1) = 2^{n+1} - 2$$

When multiplying two limbs of n-bit together, we potentially obtain a $2n$-bit value. We can see that by multiplying the maximum values again:

$$(2^n - 1) \cdot (2^n - 1) = 2^{2n} - 2 \cdot 2^n + 1 \approx 2^{2n}$$

As such, ...

Sometimes, it is easier to perform an addition or multiplication without directly reducing modulo the non-native field modulus. Reducing would allow us to get back to a normal representation of our non-native field element (e.g. four limbs of 64 bits).

Not reducing, means that we're operating on the limbs directly, and as we've seen above this means overflow! And who says overflow, also says wrapping around our circuit modulus, which is a big no no.

For this reason, the code tracks the size of the limbs (including overflows that might have occurred) and ensures that the next resulting limbs will have manageable overflow (meaning that they won't overwrap the circuit modulus).

The addition is quite simple to understand, so let's focus on the multiplication here. If we were to perform a multiplication between two integer elements and their limbs, we would compute something like this:

$$a \cdot b = (\sum_i a_i \cdot 2^{ti})(\sum_i b_i \cdot 2^{ti}) = \sum_{i,j} a_i \cdot b_j \cdot 2^{(i+j)t}$$

If we want to imagine three limbs, we would get something like this:

$$\begin{gathered} a_0 + b_0 + \\ (a_0 \cdot b_1 + a_1 \cdot b_0) \cdot 2^t + \\ (a_0 \cdot b_2 + a_1 \cdot b_1 + a_2 \cdot b_0) \cdot 2^{2t} + \end{gathered}$$

$$(a_1 \cdot b_2 + a_2 \cdot b_1) \cdot 2^{3t} + (a_2 \cdot b_2 \cdot) 2^{4t}$$

Which gives us a result scattered across 5 limbs, with the middle limb being the largest.

If we generalize that, we obtain a result of $l_0 + l_1 - 1$ limbs, if l_0 (resp. l_1) is the number of limbs of a (resp. b). With the middle limb being of size $2t + o_1 + o_2 + \lceil \log_2(n) \rceil$ where n is the number of terms in the largest limb (that middle limb).

We can see that by taking the maximum values again. We have n' pairwise multiplication of limbs where a limbs have an overflow of o_1 and b limbs have an overflow of o_2. Thus we have:

$$\begin{aligned} &(2^{t+o_1} - 1)(2^{t+o_2} - 1) \cdot n \\ &= (2^{2t+o_1+o_2} - 2^{t+o_1} - 2^{t+o_2} + 1) \cdot n \\ &< (2^{2t+o_1+o_2} \cdot 2^{\lceil \log_2(n) \rceil} \end{aligned}$$

> *"Note: Interestingly, the overflows are tracked element-wise, and not limb-wise. Not sure why I find that interesting, maybe it wouldn't make sense to track that at the limb level because we have to care about the largest limb to know if we need to reduce the whole element (and its limbs) or not."*

> *"Note: we don't care about overflow of the total value, because we never really act on the total value. The overflow only ever impacts the operation between two limbs and that's it!"*

Elliptic curve gadgets

Several elliptic curves and their operations (including pairings for pairing-friendly curves) are implemented on top of different circuit fields.

When a curve's base field is the same as the circuit field, then the curve is implemented "natively", meaning that its coordinates are represented directly as field elements in the circuit field, and operations are implemented using the default arithmetic supported by the circuit field.

On the other hand, when a curve's base field differs from the circuit field, then the emulated field types are used (see **Non-native arithmetic for fields**) to emulate the coordinates of a point, increasing the number of constraints in elliptic curve operations implemented on top.

There are three curves implemented via emulated field types **BLS12-381**, **BN254**, and **BW6-761**.

For the native curves, **BLS12-377** is implemented natively on top of BW6-761 and **BLS24-315** is implemented natively on top of **BW6_633**.

In addition, native twisted Edward curves are defined for every circuit field. For example, **Baby Jubjub** for BN254.

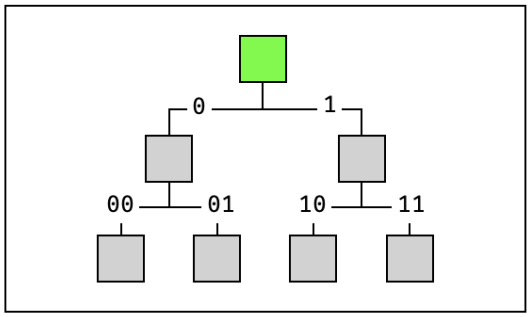
Multiplexer gadgets

The multiplexer libraries implement dynamic indexing in data structures. Since we're in a circuit, dynamic indexing means that lookup algorithms are at best linear, as they have to handle every value potentially being checked out.

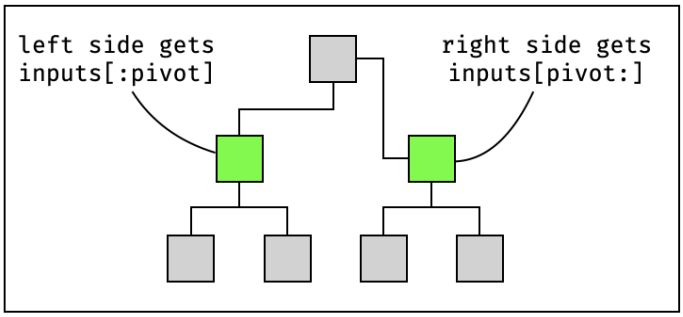
This makes implementing array lookups, slicing arrays, associated arrays, quite complicated and costly in practice. This section explains how the implementations work.

N-to-1 multiplexers in `mux` and `multiplexer`

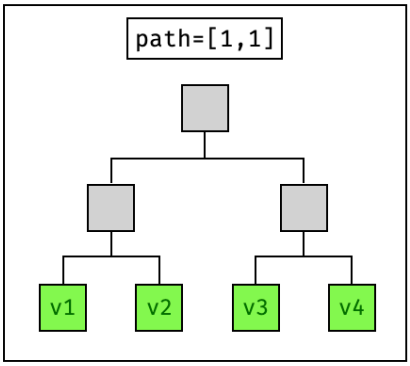
There are two ways that multiplexers are implemented. The first way is used when the array of value is a power of 2, allowing the use of a binary tree:



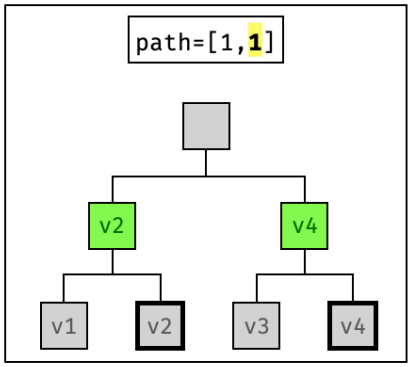
first the function goes down recursively



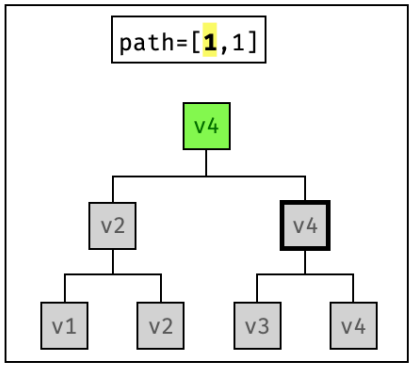
pivot is used to split the inputs in two



once it reaches the bottom, leaf values are returned



then left values are propagated up, **unless** the bit indicates otherwise



finally the correct value is returned

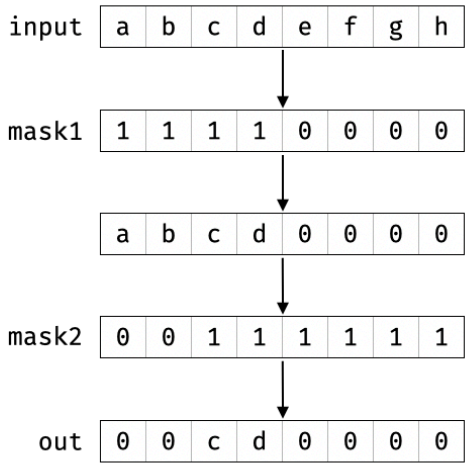
The second way is to simply create a *mask*: an array of bits $\vec{b}$ where a single bit is set to 1 where a value must be retrieved. The mask is then verified in circuit to be a well-formed *hot vector*, i.e. that it only has a single bit set in the correct position and surrounded by 0s). The value v is finally obtained by summing the dot products of the mask and the values $\vec{v}$:

$$v = \sum_i b_i \cdot v_i$$

From these two ways of constructing N-to-1 multiplexers, arrays and associated arrays are implemented as simple wrappers around these constructions.

N-to-n multiplexers in `slices`

The slice implementation allows a caller to provide an interval within an array, and nullify anything not in that interval. The implementing uses mask to nullify the right side, and then the left side, of that interval, as picture below:



The mask is used so that values are copied from the input only when the matching bit within the mask is 1, and not copied when the matching bit within the mask is 0. In pseudo-code:

```
def(input_, mask1, mask2):
    for i in range(0, len(out)):
        # mask1 = [1, 1, 1, 1, 0, 0, 0, 0]
        #                      ^
        #                     end
        out[i] = input_[i] * mask1[i]

    for i in range(0, len(out)):
        # mask 2 = [0, 0, 1, 1, 1, 1, 1, 1]
        #                  ^
        #                start
        out[i] = out[i] * mask2[i]

    # at this point out only contains the [start, end] range of input:
    # out = [0, 0, _, _, 0, 0, 0, 0]
    return out
```

The mask is passed as a hint, and the following function that constrains the correctness of the mask is implemented in the gadget:

```
def verify_mask(out, pos, start_val, end_val):
    # pos = 0 means that everything will be end_val
    if pos != 0:
        assert out[0] == start_val

    # pos = -1 means that everything will be start_val
    if pos != len(out) - 1:
        assert out[-1] == end_val

    # ensure the following:
    #
    # [start_val, start_val, end_val, end_val, end_val]
    #                        ^^^^^^^
    #                          pos
    for i in range(1, len(out)):
        if i != pos:
            assert out[i] = out[i-1]
```

Range checks implementation

Range checks are gadgets that check if a value v belongs to some interval $[a, b]$. The lower-level APIs that are exposed in gnark's std library allow developers to check specifically that a value is in the range $[0, 2^n)$ for some n. In other words, a range check enforces that a value v is n bits.

These range checks are implemented in two ways: either by verifiably decomposing the value into an n-bit array-- where verifiably means that $\sum_i b_i 2^i = v$ is added as a constraint in the circuit-- or by using a table lookup.

To range check values with a table lookup, the idea is pretty simple: create a table of all the values between 0 and some power of 2 (decided dynamically at compile time based on the bit sizes requested by the circuit range checks). Then cut a value v in limbs of size that power of 2 (verifiably, as explained above), and check that each limb is in the table (via a table lookup argument is out of scope for this document).

As the last limb might not be a perfect power of 2, its value is shifted to the left to fit the max value of the table.

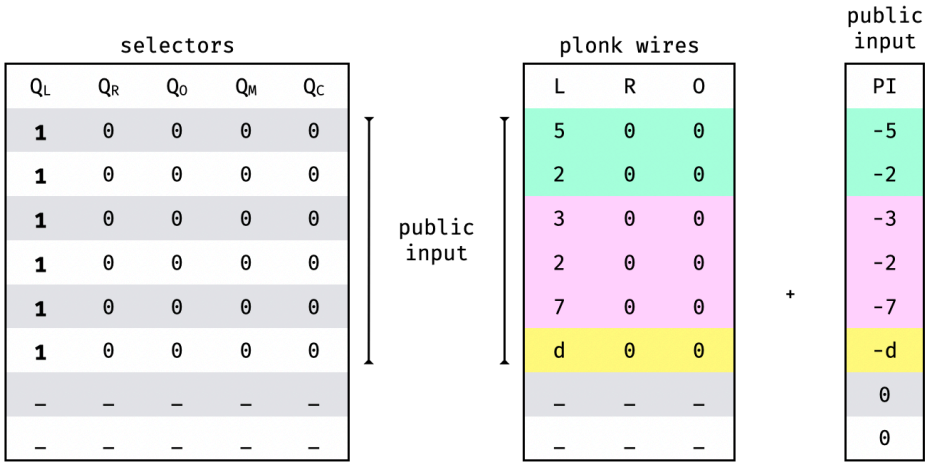
For example, if we need to check that a value v is 9 bits using a table that gathers elements from 0 to $2^4 - 1$, then we will need three limbs of 4 bits, with the last limb being 1 bit. To check the last limb, the value is shifted by 3 bits to the left.

Out-of-circuit Fiat-Shamir with Plonk

The computation of expensive functions in circuits are quite challenging, or even self-defeating in some cases. For this reason, Linea designed and implemented a scheme to delegate a specific computation: deriving randomness in a circuit. This design was introduced in the Fiat-Shamir computation of GKR statements in **Recursion over Public-Coin Interactive Proof Systems; Faster Hash Verification** (BSB22), but is used in the context of non-native arithmetic in the code we looked at.

To understand how the scheme works, first let's take a simpler scenario and solution. Imagine that at some points in a circuit we want to compute the hash of some values, `` `HASH(a, b, c)` ``, but that the hash function is very expensive to compute.

One way to solve that, is to just hint the result of the computation `` `d = HASH(a, b, c)` `` in the circuit (as the prover), and then expose `` `a, b, c, d` `` as public inputs. (And of course, make sure that the circuit wires these new public input values to their associated witness values in the circuit.) We illustrate this in the following diagram (where the pink rows are the new public inputs that are exposed, and `` `d` `` is the result of the hash function):



The verifier can then verify that the digest `` `d` `` was computed correctly before interpolating the public input vector. (Or better, produce `` `d` `` themselves and insert it into the public input vector to avoid forgetting to check it. This works as plonk checks that the following constraint checks out as part of the protocol (if you imagine that other wires and gates are not enabled on the public input rows):

$$\mathrm{L}(\mathrm{x}) - \mathrm{PI}(\mathrm{x}) = 0$$

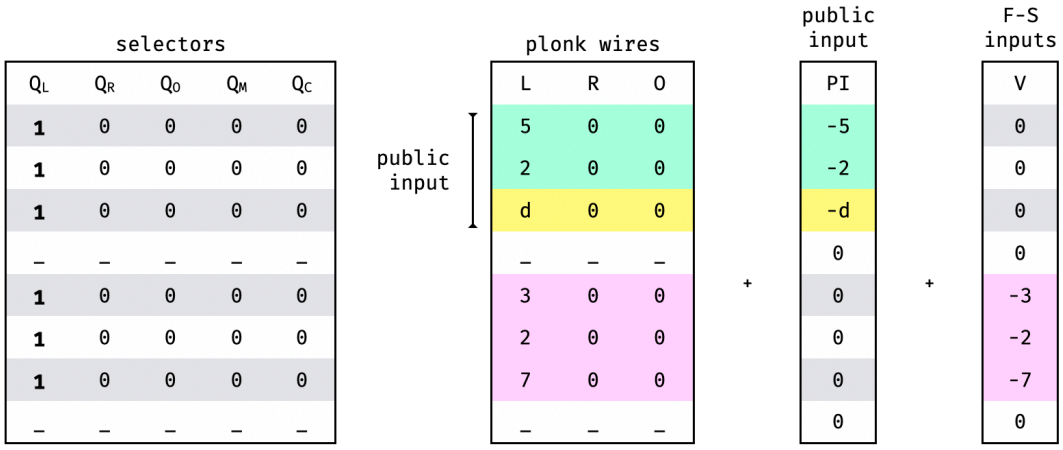
But at this point, the witness values that take part in the hash computation are exposed in the public input, and are thus leaked...

Can we hide these values to the verifier? Notice that for Fiat-Shamir use cases, we do not need to compute exactly the `` `HASH` `` function, but rather we need to generate random values based on the inputs. That's right, it does not matter

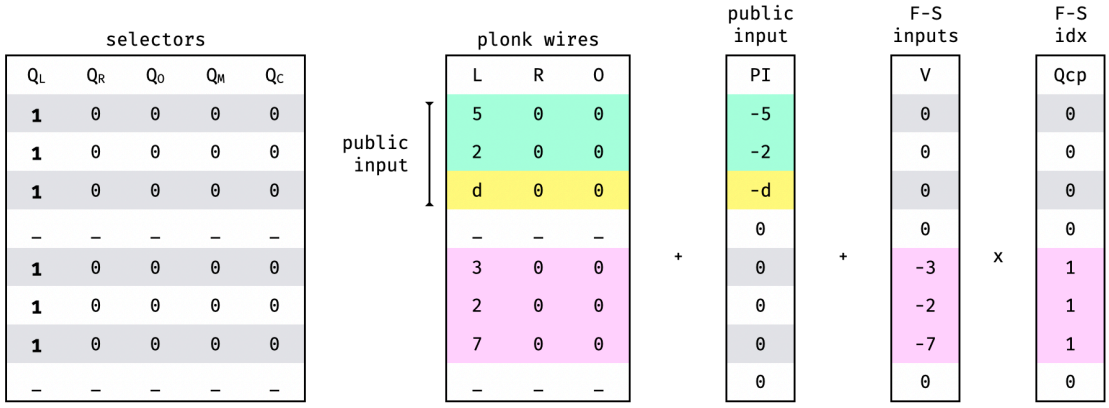
how we generate these random values as long as they are based on the private inputs. As such, we can relax our requirements and we can do something like this instead: `d = HASH(commit(a, b, c))`.

As the challenge is used for Fiat-Shamir, we can leave it in the public input, but now the inputs can be kept in the witness. Although, they are kept in rows that resemble public input rows, in that the left wire is enabled and no gate is selected.

Then, a hiding commitment of the values is produced by the prover and sent to the verifier, as we show in this diagram:



We are almost here, there's still one more problem: we need to prevent the prover from using this new commitment to alter random witness values. To do this, the verifier key includes one more selector vector that dictates where this new commitment can apply:



Which ends up producing a constraint that looks like this (if again, we ignore the gates and wires not enabled in these rows):

$$\mathrm{L}(\mathrm{x}) - \mathrm{PI}(\mathrm{x}) - \mathrm{V}\,(\mathrm{x}) \cdot \mathrm{Qcp}(\mathrm{x}) = 0$$

Note that the implementation we looked at was generalized to work with n Fiat-Shamir instances by having n commitments $[\mathrm{V_i}(\mathrm{x})]$ and n committed selectors $[\mathrm{Qcp_i}(\mathrm{x})]$, as well as n reserved rows of public inputs for the n

digests.

On top of that, they evaluate commitments to the selectors $[\mathrm{Qcp_i(x)}]$ during the linearization phase of Plonk, so as not to care about the randomization of the committed inputs to Fiat-Shamir, although the commitment is still randomized by randomizing entries of the Qcp vector in rows that are not activated.

In-Circuit KZG Verification

The in-circuit KZG implementation in `gnark/std/commitments/kzg` closely follows the one in the gnark-crypto library.

Single-Point Opening Proof Verification

For claimed value $v = f(u)$ of polynomial $f(X)$ at point u, KZG polynomial commitment verification is implemented as a pairing check:

$$e(v \cdot G_1 - [f(\alpha)]G_1 - u \cdot [H(\alpha)]G_1, G_2) \cdot e([H(\alpha)]G_1, [\alpha]G_2) \stackrel{?}{=} 1$$

The *commitment* of $f(X)$ is a $\mathbb{G}_1$ element $[f(\alpha)]G_1$.

The *evaluation point* is $u \in \mathbb{F}_r$ — an element of the scalar field of $\mathbb{G}_1$.

The *opening proof* is just 2 values:

- $[H(\alpha)]G_1$ — commitment of the *quotient polynomial* $H(X)$,
- $v = f(u) \in \mathbb{F}_r$ — claimed value of $f(X)$ at evaluation point u.

The *verification key* consists of

- $G_2, [\alpha]G_2$ — generator of group $\mathbb{G}_2$ scaled to 1 and α — toxic secret (supposed to be) destroyed after the generation of SRS (powers of α multiplied with G_1 and G_2).
- G_1 — generator of group $\mathbb{G}_1$.

The above check is equivalent to checking that

$$e([f(u) - f(\alpha)]G_1, G_2) = e([(u - \alpha) \cdot H(\alpha)]G_1, G_2)$$

Which it should if there exists the quotient polynomial $H(X)$, defined as

$$H(X) = \frac{f(u) - f(X)}{u - X}$$

The prover provides $[H(\alpha)]G_1$ as part of the proof, which is highly unlikely (with probability $1/|\mathbb{F}_r|$) if the prover doesn't know the polynomial $H(X)$.

Fold Batch Opening Proof at a Single Point

This method is used by the in-circuit Plonk verifier to fold multiple evaluation proofs at a single point ζ.

It takes:

- a list commitments $[f_i(\alpha)]G_1$,
- evaluation point u,
- a *batch opening proof* which consists of:
  - a single quotient polynomial commitment $[H(\alpha)]G_1$,
  - a list of values of the committed polynomials at the evaluation point $v_i = f_i(u)$.

And produces:

- folded commitment $[f]G_1$, and
- folded opening proof $([H(\alpha)]G_1, v)$ – here, the quotient commitment is simply passed through from the input, and v is a produced folded evaluation.

A random challenge λ is obtained by hashing all the input data, except the quotient commitment $[H(\alpha)]G_1$ because it is computed by the prover at a later stage and is itself dependent on λ.

The logic from here is straightforward. Folded commitment:

$$[f]G_1 = \sum_{i=0}^{n-1} \lambda^i \cdot [f_i(\alpha)]G_1$$

Folded evaluation:

$$v = \sum_{i=0}^{n-1} \lambda^i \cdot f_i(u_i)$$

Multi-Point Batch Verification

This method is used by the in-circuit Plonk verifier instead of verification steps 11-12 in the Plonk paper.

Multi-point batch verification of KZG commitments and opening proofs takes a list of triplets (for $i \in [0, n-1]$), of:

- polynomial commitment $[f_i(\alpha)]G_1$
- evaluation point $u_i \in \mathbb{F}_r$
- opening proof $([H_i(\alpha)]G_1, v_i = f_i(u_i))$

All of them are checked against a single verification key $(G_1, G_2, [\alpha]G_2)$.

The final check boils down to a pairing check:

$$e(v \cdot G_1 - [f]G_1 - [uH]G_1, G_2) \cdot e([H]G_1, [\alpha]G_2) \stackrel{?}{=} 1$$

It looks similar the one for a single KZG proof verification. We will see in a second, how its components are obtained.

A random value λ is produced by hashing all the input data. Then it's used to produce folded values for the final pairing check.

Folded quotients:

$$[\mathrm{H}]\mathrm{G}_1 = \sum_{\mathrm{i}=0}^{\mathrm{n}-1} \lambda^{\mathrm{i}} \cdot [\mathrm{H}_{\mathrm{i}}(\alpha)]\mathrm{G}_1$$

Folded products of evaluation points and quotients:

$$[\mathrm{uH}]\mathrm{G}_1 = \sum_{\mathrm{i}=0}^{\mathrm{n}-1} \lambda^{\mathrm{i}} \cdot \mathrm{u}_{\mathrm{i}} \cdot [\mathrm{H}_{\mathrm{i}}(\alpha)]\mathrm{G}_1$$

Folded evaluations:

$$\mathrm{v} = \sum_{\mathrm{i}=0}^{\mathrm{n}-1} \lambda^{\mathrm{i}} \cdot \mathrm{f}_{\mathrm{i}}(\mathrm{u}_{\mathrm{i}})$$

Folded commitments:

$$[\mathrm{f}]\mathrm{G}_1 = \sum_{\mathrm{i}=0}^{\mathrm{n}-1} \lambda^{\mathrm{i}} \cdot [\mathrm{f}_{\mathrm{i}}(\alpha)]\mathrm{G}_1$$

Findings

Below are listed the findings found during the engagement. High severity findings can be seen as so-called "priority 0" issues that need fixing (potentially urgently). Medium severity findings are most often serious findings that have less impact (or are harder to exploit) than high-severity findings. Low severity findings are most often exploitable in contrived scenarios, if at all, but still warrant reflection. Findings marked as informational are general comments that did not fit any of the other criteria.

ID	COMPONENT	NAME	RISK
00	std/multicommit/nativecommit.go	MultiCommitter Scheme May Converge To Using Constant Challenge 1 After Many Calls	High
01	std/math/bitslice/partition.go	Partition Function Is Not Sound	High
02	std/algebra/native	GLV Decomposition In Scalar Multiplication In Native Field Is Not Sound	High
03	std/math/uints	Conversion Of Uint Type Is Not Sound	High
04	std/math/emulated	Non-Cryptographic Hash Function Could Lead To Underconstrained Foreign Elements	Medium
05	std/math/emulated/field_assert.go	Incorrect Condition May Lead To Overflow Native Field In IsZero Function	Medium

ID	COMPONENT	NAME	RISK
06	std/hash/mimc	MiMC Implementation Is Vulnerable To Length-Extension Attacks	Medium
07	std/math/emulated/custommod.go	Missing Constraints May Lead To Underflow In modSub Function	Medium
08	std/math/emulated	Reduce Function Is Misleading And Could Lead To Soundness and Completeness Issues	Medium
09	std/math/bits/naf.go	Missing Constraints In ToNAF Function	Medium
0a	std/recursion/wrapped_hash.go	wrapped_hash Is Not Collision Resistant With Input Of Different Length	Low
0b	std/math/emulated	Field API Is A Leaky Abstraction	Informational

00 - MultiCommitter Scheme May Converge To Using Constant Challenge 1 After Many Calls

- std/multicommit/nativecommit.go

High

Description. `multicommitter` collects variables from multiple functions and commits them with underlying `Committer` to get the initial commitment c. Then it derives the commitment for all the functions by continuously squaring: $c \leftarrow c^2$.

```
func (mct *multicommitter) commitAndCall(api frontend.API) error {
    ...
    cmt, err := commiter.Commit(mct.vars...)
    ...
    for i := 1; i < len(mct.cbs); i++ {
        // cmt <-- cmt * cmt
        cmt = api.Mul(cmt, cmt)
        if err := mct.cbs[i](api, cmt); err != nil {
            return fmt.Errorf("callback %d: %w", i, err)
        }
    }
    return nil
}
```

The way it derives the commitement is not binding. This will make the fiat-shamir transformation not sound.

One observation is that: in prime field $\mathbb{F}_p$, for some c the following sequence will converge to 1:

$$c, c^2, c^4, c^8, \ldots, c^{2^i}, c^{2^{i+1}}, \ldots$$

This is because if $c^{2^i} \equiv 1 \mod p$ then $c^{2^{i+1}} \equiv 1 \mod p$.

This means if there are many c in $\mathbb{F}_p$ that makes the sequence converge to 1, the attacker can easily find two different inputs both committed to 1. This will make `multicommitter` lose binding.

Now let's see in $\mathbb{F}_p$ how many c will eventually converge to 1 in the sequence: $c, c^2, c^4, c^8, \ldots, c^{2^i}, c^{2^{i+1}}, \ldots$

Denote g as the generator in $\mathbb{F}_p$. Then $g^{p-1} \equiv 1 \mod p$. Then $c = g^k$ for some k. We have:

$$c^{2^n} = (g^k)^{2^n} = g^{k \cdot 2^n}$$

Suppose the sequence converge to 1 at n:

$$c^{2^n} \equiv g^{k \cdot 2^n} \equiv 1 \mod p$$

This means $p - 1$ divides $k \cdot 2^n$:

$$(p - 1) \mid k \cdot 2^n$$

Factorize $p - 1$ as $a \cdot 2^m$, with m being the largest possible value. If $a \mid k$, then $a \cdot 2^m \mid k \cdot 2^n$ for $m \leq n$, implying that $(p - 1) \mid k \cdot 2^n$ and $g^{k \cdot 2^n} \equiv 1 \mod p$. This means the sequence converges to 1 if and only if $a \mid k$. Consequently, if $a \mid k$, the sequence converges to 1 in at most m steps.

In $\mathbb{F}_p$, there are approximately p/a elements divisible by a. If one randomly chooses an element in $\mathbb{F}_p$, the probability that the element is divisible by a (i.e., the sequence converges to 1) is $1/a$.

For some prime fields, $1/a$ is non-negligible. For instance, gnark supports the **Stark Curve**, where $p = 2^{251} + 17 \cdot 2^{192} + 1$. Consequently, $p - 1 = 2^{192} \cdot (2^{59} + 17)$, which means $a = 2^{59} + 17$ and $m = 192$. Suppose the circuit makes 192 calls to the `multicommitter`. Then, an attacker could find two different inputs, both committed to 1 in the 192nd call, in $O(2^{59} + 17)$ attempts.

This indicates that in some prime fields, the `multicommitter` is not binding.

Recommendation: Avoid using a squaring sequence. Consider using c^i to derive the sequence.

Client Response. https://github.com/Consensys/gnark/pull/1212

01 - Partition Function Is Not Sound

- std/math/bitslice/partition.go

High

Description. The `Partition()` function partitions a value `v` into two parts (a lower part and an upper part) at a specific bit offset. In order to be sound, the following should hold:

1. $\mathrm{v} = \mathrm{lower} + 2^{\mathrm{split}} \times \mathrm{upper}$.
2. $\mathrm{lower} < 2^{\mathrm{split}}$.
3. $\mathrm{upper} < 2^{\mathrm{split}'}$, where $\mathrm{split}' = \mathrm{nbScalar} - \mathrm{split}$.

Unfortunately, in the implementation the upper part is incorrectly constrained to be $\mathrm{upper} < 2^{\mathrm{nbScalar}}$. Then any lower and upper pairs that satisfy the first two constraints will pass the check. Thus, the implementation is not sound.

```
func Partition(api frontend.API, v frontend.Variable, split uint, opts ...Option) (lower, upper
frontend.Variable) {
    ...
    upperBound := api.Compiler().FieldBitLen()
    if opt.digits > 0 {
        upperBound = opt.digits
    }
    // incorrect constraint here
    rh.Check(upper, upperBound)
    rh.Check(lower, int(split))

    m := big.NewInt(1)
    m.Lsh(m, split)
    composed := api.Add(lower, api.Mul(upper, m))
    api.AssertIsEqual(composed, v)
    return
}
```

Besides, the function supports input `nbDigits` to constrain the bit number of `v`. If nbDigits is not specified (`nbDigits = api.Compiler().FieldBitLen()`), there can be two ways to split the value. For example, to split 1, we can make $\mathrm{lower} + 2^{\mathrm{split}} \times \mathrm{upper}$ equals to 1 or $\mathrm{p} + 1$, where p is the field modulus. If nbDigits is specified and `nbDigits > api.Compiler().FieldBitLen()` holds, there will be multiple ways to split the value.

Recommendation. Constrain the upper part to be of the correct size:

```
rh.Check(upper, upperBound - int(split))
```

In addition, add an assertion in the function to ensure that

```
nbDigits < api.Compiler().FieldBitLen()
```

Client Response. https://github.com/Consensys/gnark/pull/1165

02 - GLV Decomposition In Scalar Multiplication In Native Field Is Not Sound

- std/algebra/native

High

Description. gnark uses GLV decomposition to speedup scalar multiplication in native field. To compute `P=[s]Q`, one key step is to decompose `s` into two smaller parts `s1` and `s2` such that `s1 + λ * s2 == s mod r`, where `λ` is fixed value and `r` is the scalar field. To do that, gnark uses a hint and obtains `s1` and `s2`, it then check if the equation holds:

```
// s1 + λ * s2 == s mod r
api.AssertIsEqual(
    api.Add(s1, api.Mul(s2, cc.lambda)),
    api.Add(s, api.Mul(cc.fr, m)),
)
```

This check is intended to ensure there exists `m` such that `s1 + λ * s2 == s + m*r`. However, this check is performed in the native field and may incur overflow. It's actually checking that there exists `m` and `n` such that `s1 + λ * s2 == s + m*r + n*baseField(i.e. nativeField)`. As `m` and `n` can be arbitrary values, there can be multiple ways to decompose `s`. The decomposition check is not sound.

Recommendation. The decomposition check should be done in the scalar field instead of the base field (i.e. native field). Consider performing the check using an emulated field types.

Client Response. https://github.com/Consensys/gnark/pull/1167

03 - Conversion Of Uint Type Is Not Sound

- std/math/uints

High

Description. Gnark's standard library defines some gadgets to emulate uint32 and uint64 types.

An important aspect of these interfaces is to ensure that the conversion from native circuit field elements and the uint types is sound. This is done through the `ValueOf` and `ToValue` functions. Here is the `ValueOf` function that is written generically for both the uint32 and uint64 types:

```
func (bf *BinaryField[T]) ByteValueOf(a frontend.Variable) U8 {
    bf.rchecker.Check(a, 8)
    return U8{Val: a, internal: true}
}

func (bf *BinaryField[T]) ValueOf(a frontend.Variable) T {
    var r T
    bts, err := bf.api.Compiler().NewHint(toBytes, len(r), len(r), a)
    if err != nil {
        panic(err)
    }
    // TODO: add constraint which ensures that map back to
    for i := range bts {
        r[i] = bf.ByteValueOf(bts[i])
    }
    return r
}
```

The problem with this code is that the hinted bytestring is not checked to be a correct representation of the original value. As such, it can be arbitrarily set to any value by the prover, leading to a soundness issue.

Recommendation. As with the `ToBits` function, recompose the bytestring into a value and assert that it is equal to the original one. For example:

```
    for i := range bts {
        r[i] = bf.ByteValueOf(bts[i])
    }
+    expectedValue := bf.ToValue(r)
+    bf.api.AssertIsEqual(a, expectedValue)
```

In addition, document that overflows are forbidden, as the current implementation does not allow for overflows.

Client Response. https://github.com/Consensys/gnark/pull/1139

04 - Non-Cryptographic Hash Function Could Lead To Underconstrained Foreign Elements

- std/math/emulated

Medium

Description. The `enforceWidthConditional` is used in several places in the circuit to constrain that the limbs of foreign field elements are correctly formed (i.e. that they are of the correct bitlength).

The function relies on a hash function `HashCode()` to mark if a linear combination has been seen and constrained before:

```
func (f *Field[T]) enforceWidthConditional(a *Element[T]) (didConstrain bool) {
    // TRUNCATED...
    for i := range a.Limbs {
        // TRUNCATED...
        if vv, ok := a.Limbs[i].(interface{ HashCode() uint64 }); ok {
            // okay, this is a canonical variable and it has a hashcode. We use
            // it to see if the limb is already constrained.
            h := vv.HashCode()
            if _, ok := f.constrainedLimbs[h]; !ok {
                // we found a limb which hasn't yet been constrained. This means
                // that we should enforce width for the whole element. But we
                // still iterate over all limbs just to mark them in the table.
                didConstrain = true
                f.constrainedLimbs[h] = struct{}{}
            }
        } else {
            // we have no way of knowing if the limb has been constrained. To be
            // on the safe side constrain the whole element again.
            didConstrain = true
        }
    }
    if didConstrain {
        f.enforceWidth(a, true)
    }
    return
}
```

This hash function is unfortunately not cryptographically secure, as the following code shows:

```
// -- gnark/frontend/internal/expr/linear_expression.go --
func (l LinearExpression) HashCode() uint64 {
    h := uint64(17)
```

```
    for _, val := range l {
        h = h*23 + val.HashCode() // TODO @gbotrel revisit
    }
    return h
}

// -- gnark/frontend/internal/expr/term.go --
func (t Term) HashCode() uint64 {
    return t.Coeff[0]*29 + uint64(t.VID<<12)
}
```

As such, it might be possible for a malicious circuit developer to create a circuit in which two different circuit variables collide and produce the same digest, leading to only one of them being properly constrained.

Note that other parts of the gnark library use the `HashCode()` function, for example, `MarkBoolean` and `IsBoolean`.

Recommendation. We recommend using a cryptographically secure hash function that would have both collision resistance and pre-image resistance, in order to ensure that circuit builders cannot produce stealthily underconstrained circuits.

Client Response. https://github.com/Consensys/gnark/pull/1197

05 - Incorrect Condition May Lead To Overflow Native Field In IsZero Function

● std/math/emulated/field_assert.go

Medium

Description. `IsZero` returns a boolean indicating if all the limbs of the element are exactly zero. The safe (slow) way is to check if each of the limbs is zero one-by-one. The fast way is to check if the sum of all the limbs is zero, provided it won't cause an overflow in the native field. However, the condition of judging if it will cause overflow is flipped.

```
// IsZero returns a boolean indicating if the element is strictly zero.
func (f *Field[T]) IsZero(a *Element[T]) frontend.Variable {
    ...
    // every addition adds a bit to the overflow
    totalOverflow := len(ca.Limbs) - 1
    // the condition here is incorrect.
    if totalOverflow < int(f.maxOverflow()) {
        // safe way: check if each limb is zero, one by one.
        res := f.api.IsZero(ca.Limbs[0])
        for i := 1; i < len(ca.Limbs); i++ {
            res = f.api.Mul(res, f.api.IsZero(ca.Limbs[i]))
        }
        return res
    }
    // fast way: check if the sum of all the limbs is zero.
    limbSum := ca.Limbs[0]
    for i := 1; i < len(ca.Limbs); i++ {
        limbSum = f.api.Add(limbSum, ca.Limbs[i])
    }
    return f.api.IsZero(limbSum)
}
```

Here, if `totalOverflow >= int(f.maxOverflow())` is true, it means the sum will cause an overflow and it should use the safe way. However, The code indirectly routes it to the fast way, which may lead to overflow. A malicious prover can prove non-zero element to be zero utilizing the overflow.

In current gnark's parameter setting, `totalOverflow < int(f.maxOverflow())` is always true. This means it will always route to the safe way. Therefore it only leads to downgraded efficiency. However, it can be problematic for the downstream users if they have different parameter settings.

Recommendation. Change it to the right condition. For example:

```
func (f *Field[T]) IsZero(a *Element[T]) frontend.Variable {
    ...
```

```
    if totalOverflow > int(f.maxOverflow())
    ...
}
```

Client Response. https://github.com/Consensys/gnark/pull/1145

06 - MiMC Implementation Is Vulnerable To Length-Extension Attacks

● std/hash/mimc

Medium

Description. Gnark's standard library implements the **MiMC hash function** using the **Miyaguchi–Preneel** construction. The Miyaguchi–Preneel construction is a known way to turn a block cipher into a compression function, which can then be used to build a hash function.

Such an instantiation is well-studied and known to be secure against most attacks, except **length-extension attacks**. Length-extension attacks occur when a secret is used in the hash function to produce a keyed hash, for example, if one produces a digest as `` `hash(secret || public_data)` `` where `` `||` `` is a concatenation, someone else could pick up where the hashing was left off and produce a new digest `` `hash(secret || public_data || more_data)` `` without knowing the secret. Fundamentally, this is because the digest produced by the algorithm is the internal state of the hash function, which can be reused without issue to continue hashing.

As such, one could imagine an innocent developer producing a circuit where `` `data` `` and the digest are made public (through public inputs, for example) and then used in another protocol and circuit to produce a different keyed-hash on some related data (as explained above). For example:

```
type Circuit1 struct {
    Key frontend.Variable `gnark:",secret"`
    Data [1]frontend.Variable `gnark:",public"`
    Expected frontend.Variable `gnark:",public"`
}

type Circuit2 struct {
    Key frontend.Variable `gnark:",secret"`
    Data [2]frontend.Variable `gnark:",public"`
    Expected frontend.Variable `gnark:",public"`
}

func (c *CircuitN) Define(api frontend.API) error {
    h, err := mimc.New(api)
    if err != nil { return err }
    h.Write(c.Key)
    h.Write(c.Data[:]...)
    res := h.Sum()
    api.AssertIsEqual(res, c.Expected)
}
```

In addition, the **MiMC website's FAQ** also states:

> *"In our original paper we propose to instantiate a hash function via plugging the MiMC permutation into a Sponge construction. The reason back then was security analysis. A mode like Miyaguchi-Preneel makes it harder, and in 2016 we did not feel confident in proposing this. In the meanwhile we did more security analysis, improving our understanding and confidence, but our recommendation remains unchanged."*

Note that we could not find usages of this API being used with secret data, but the API itself remains exposed to end users and does not forbid the caller to use it with secret data.

Recommendation. Document the API to make it clear that it should not be used with secret data.

Client Response. https://github.com/Consensys/gnark/pull/1198

07 - Missing Constraints May Lead To Underflow In modSub Function

- std/math/emulated/custommod.go

Medium

Description. `modSub` function performs sub operation on emulated `Element` over given modulus `p`. To calculate `(a - b) % p`, gnark uses a padding `pad` to avoid any possible underflow and overflow in the native field. Then it calculates `a[i] + pad[i] - b[i]` for all `i` and recomposes the result.

The `pad` should satisfies:

1. correctness: `pad % p = 0`.
2. avoid underflow: `pad[i] >= b[i]` for all i.
3. avoid overflow `pad[i] + a[i] < native_field` for all i.

Unfortunately, `pad` is given by a hint function and misses a constraint to enforce condition 2. A malicious prover may provide a `pad` that satisfies 1 and 3, but causes underflow when calculating `a[i] + p[i] - b[i]`. In this way the `modSub` function will return incorrect results.

```
func (f *Field[T]) computeSubPaddingHint(overflow uint, nbLimbs uint, modulus *Element[T])
*Element[T] {
    var fp T
    inputs := []frontend.Variable{fp.NbLimbs(), fp.BitsPerLimb(), overflow, nbLimbs}
    inputs = append(inputs, modulus.Limbs...)
    res, err := f.api.NewHint(subPaddingHint, int(nbLimbs), inputs...)
    if err != nil {
        panic(fmt.Sprintf("sub padding hint: %v", err))
    }
    for i := range res {
        // enforce condition 3
        f.checker.Check(res[i], int(fp.BitsPerLimb()+overflow+1))
    }
    padding := f.newInternalElement(res, fp.BitsPerLimb()+overflow+1)
    // enforce condition 1
    f.checkZero(padding, modulus)
    return padding
}
```

Another issue is incorrect `overflow` setting. The `overflow` of `padding` should be `overflow+1` instead of `fp.BitsPerLimb()+overflow+1`. In the `modSub` function, `nextOverflow` is calculated as `nextOverflow := max(b.overflow+1, a.overflow) + 1`. This will be an issue if `nextOverflow > f.maxOf()`.

Recommendation. Add constraint to enforce `pad[i] >= b[i]` for all i. Correctly enforce the overflow setting.

Client Response. https://github.com/Consensys/gnark/pull/1200

08 - Reduce Function Is Misleading And Could Lead To Soundness and Completeness Issues

- std/math/emulated

Medium

Description. Non-native field elements are represented as `Element` variables in gnark circuits. Such variables can be provided by the prover and are checked to have limbs of the correct bitlength (as explained in the introduction of this report). Let's call this bitlength n.

There are two kinds of overflows that can happen with these values:

1. *bit overflows*: when the limbs comprising the `Element` variable now need to track larger bitstrings (e.g. $n+1$-bit limbs) because an overflow might have happened. This also means that the value being tracked might potentially be larger than the modulus.
2. *modulus overflow with no bit overflow*: the value is larger than the modulus, but its limbs did not have bit overflows. This is the kind of overflow that we are interested in.

> *"Note that `Element` variables that are known to have no bit overflows are marked as `internal`."*

A `Reduce` function is provided which only performs a modular reduction if it detects no bit overflow. This means that it is possible that the function returns a value larger than the modulus in the case of *modulus overflow with no bit overflow*. For example, it could return $f+1$ with f the foreign modulus.

This behavior implies that the codebase makes the following general assumptions:

1. Either functions don't care about *modulus overflow with no bit overflow*, as they can deal with it. This is the case of many of the non-native arithmetic operations like `Add` and `Mul`.
2. Or functions care, and they will call the `AssertIsInRange()` function to prevent malicious provers from using values that are larger than the modulus.

Both of these assumptions are not necessarily true.

The first assumption doesn't hold for the `ToBits` and `Exp` functions. For example, `ToBits(f+1)` and `ToBits(1)` will return different bits. Looking at `ToBits` first, we can see that the comment below is misleading because the `Reduce` function won't reduce a modulus overflow with no bit overflow (and thus won't generate a canonical representation of Element as a bitstring).

```
// ToBits returns the bit representation of the Element in little-endian (LSB
// first) order. The returned bits are constrained to be 0-1. The number of
// returned bits is nbLimbs*nbBits+overflow. To obtain the bits of the canonical
// representation of Element, reduce Element first and take less significant
```

```
// bits corresponding to the bitwidth of the emulated modulus.
func (f *Field[T]) ToBits(a *Element[T]) []frontend.Variable {
    f.enforceWidthConditional(a)
    ...
```

The `Exp` function uses `ToBits` to generate the bits and perform exponentiation by squaring. As `ToBits` may generate different bits, `Exp` will generate incorrect results.

```
// Exp computes base^exp modulo the field order. The returned Element has default
// number of limbs and zero overflow.
func (f *Field[T]) Exp(base, exp *Element[T]) *Element[T] {
    expBts := f.ToBits(exp)
    n := len(expBts)
    res := f.Select(expBts[0], base, f.One())
    base = f.Mul(base, base)
    for i := 1; i < n-1; i++ {
        res = f.Select(expBts[i], f.Mul(base, res), res)
        base = f.Mul(base, base)
    }
    res = f.Select(expBts[n-1], f.Mul(base, res), res)
    return res
}
```

Take the below case as an example. A malicious prover can choose `f.ModMul(b, c)` to be either `l` or `f+l`. Then further choose the result of `f.ModExp(a, l)`:

```
l := f.ModMul(b, c)
res := f.ModExp(a, l)
```

Going back to our two cases, the second case (calling `AssertIsInRange` to enforce a smaller-than-modulus value) can be seen, for example, in the start of the implementation of the `IsZero` function:

```
// AssertIsInRange ensures that a is less than the emulated modulus. When we
// call [Reduce] then we only ensure that the result is width-constrained, but
// not actually less than the modulus. This means that the actual value may be
// either x or x + p. For arithmetic it is sufficient, but for binary comparison
// it is not. For binary comparison the values have both to be below the
// modulus.
func (f *Field[T]) AssertIsInRange(a *Element[T]) {
    // we omit conditional width assertion as is done in ToBits down the calling stack
    f.AssertIsLessOrEqual(a, f.modulusPrev())
}

// IsZero returns a boolean indicating if the element is strictly zero. The
// method internally reduces the element and asserts that the value is less than
// the modulus.
func (f *Field[T]) IsZero(a *Element[T]) frontend.Variable {
```

```
    ca := f.Reduce(a)
    f.AssertIsInRange(ca)
```

The issue with this pattern is that it assumes that no illegitimate `Element` value (one that would be larger than the modulus but with no bit overflow) can be produced at runtime, besides a direct hint from the prover (which in this case would not matter, as they would be shooting themselves in the foot).

While this might be true, it is not necessarily obviously true. We can expect that all field operations will increase the overflow of the element variable, which will in turn trigger the `Reduce` function and perform a modular reduction in legitimate cases (before the call to `AssertIsLessOrEqual`), but further code changes might introduce edge cases where this is not true anymore, potentially leading to completeness issues.

Recommendation. Change the name of the `Reduce` function to avoid confusing developers making use of the function. Keep track of `Element` variables that might not be "fully reduced", in the sense that they might be larger than the modulus, and enforce a modular reduction if this is the case in functions like `ToBits` and `Exp`.

Furthermore, consider taking a slightly different approach and always enforcing that `Element` values are less than the modulus if they have no bit overflow. See **Field API Is A Leaky Abstraction** for a related issue.

Client Response. https://github.com/Consensys/gnark/issues/1147

09 - Missing Constraints In ToNAF Function

- std/math/bits/naf.go

Medium

Description. `ToNAF` returns the **NAF decomposition** of given input. The non-adjacent form (NAF) of a number is a unique signed-digit representation, in which non-zero values cannot be adjacent.

gnark uses a hint and obtains the NAF decomposition **`bits`**. Then it performs checks to ensure the sum of the **`bits`** equals to the input value and each of the **`bits`** is **`-1`**, **`0`** or **`1`**.

```
// ToNAF returns the NAF decomposition of given input.
// The non-adjacent form (NAF) of a number is a unique signed-digit representation,
// in which non-zero values cannot be adjacent. For example, NAF(13) = [1, 0, -1, 0, 1].
func ToNAF(api frontend.API, v frontend.Variable, opts ...BaseConversionOption)
[]frontend.Variable {
    ...
    c := big.NewInt(1)

    bits, err := api.Compiler().NewHint(nNaf, cfg.NbDigits, v)
    if err != nil {
        panic(err)
    }

    var Σbi frontend.Variable
    Σbi = 0
    for i := 0; i < cfg.NbDigits; i++ {
        Σbi = api.Add(Σbi, api.Mul(bits[i], c))
        c.Lsh(c, 1)
        if !cfg.UnconstrainedOutputs {
            // b * (1 - b) * (1 + b) == 0
            // TODO this adds 3 constraint, not 2. Need api.Compiler().AddConstraint(...)
            b := bits[i]
            y := api.Mul(api.Sub(1, b), api.Add(1, b))
            api.AssertIsEqual(api.Mul(b, y), 0)
        }
    }

    // record the constraint Σ (2**i * b[i]) == v
    api.AssertIsEqual(Σbi, v)

    return bits
}
```

Unfortunately, the function doesn't enforce "non-zero values cannot be adjacent" in the circuit. Thus a malicious prover can generate a representation with adjacent non-zero values, which may poison the subsequent computation.

Recommendation. Add constraints to enforce "non-zero values cannot be adjacent" in the circuit.

Client Response. https://github.com/Consensys/gnark/pull/1164

0a - wrapped_hash Is Not Collision Resistant With Input Of Different Length

● std/recursion/wrapped_hash.go

Low

Description. Gnark uses wrapped_hash to wrap MiMC function for computing the challenges inside the circuit. The inputs are bits (or native field elements) and outputs are elements in the emulated field.

wrapped_hash batches the input bits and only writes a whole batch at a time to the underlying hash function. For the last batch, it will fill with zeros if the input bits don't fill up the batch. This means that wrapped_hash won't distinguish whether the trailing bits are input bits or just padding. Then, it's easy to construct two preimages of different lengths that produce the same hash:

```
func TestCollision(t *testing.T) {
    h, err := recursion.NewShort(ecc.BLS12_377.ScalarField(), ecc.BN254.ScalarField())
    if err != nil {
        panic(err)
    }

    h.Write([]byte{0x00})
    hash := h.Sum(nil)
    fmt.Printf("%x\n", hash) // 34373b65f439c874734ff51ea349327c140cde2e47a933146e6f9f2ad8eb17
    h.Reset()

    h.Write([]byte{0x00})
    h.Write([]byte{0x00})
    fmt.Printf("%x\n", h.Sum(nil)) //
34373b65f439c874734ff51ea349327c140cde2e47a933146e6f9f2ad8eb17
}
```

Currently, this is not an issue because wrapped_hash is created to generate challenges that have a fixed input length. However, it may introduce vulnerabilities when downstream users use it.

Recommendation. Add warnings in the doc to avoid misuse.

Client Response. https://github.com/Consensys/gnark/pull/1198

0b - Field API Is A Leaky Abstraction

● std/math/emulated

Informational

Description. Foreign field elements are represented as `Element` variable types in circuits.

`Element` variables have an internal field called `internal` which is set only if created through `field.NewElement` or through the foreign field methods defined in the `emulated` package. This field indicates that the limbs of the `Element` variable were constrained to be of the correct bitlength.

This `Element` type can be introduced through hints, or through private inputs to the circuit if included in the circuit struct. For example:

```
type someCircuit struct {
    SomeVariable Element[Secp256k1Fp]
}
```

When passed as private inputs, `Element` variables do not have their limbs constrained and as such do not have their `internal` field set to `true`.

For this reason, it is therefore the responsibility of functions making use of it to remember to call `enforceWidthConditional` to make sure that the field element gets properly constrained (i.e. number of limbs is correct and each limb is of the correct bitlength). For example, the `checkZero`, `Reduce`, `Select`, `Lookup2`, `Mux`, `AssertIsEqual`, `mulMod`, `reduceAndOp`, `ToBits`, `Sum`, all call the `enforceWidthConditional` function.

> *"Note: it still does not mean that the value is less than the modulus, as we discuss in Reduce Function Is Misleading And Could Lead To Soundness and Completeness Issues."*

This is a dangerous pattern that could lead to soundness issues if functions of the standard library, or user-written functions to extend the standard library, forget to call `enforceWidthConditional` on an `Element` variable.

Recommendation. We recommend ensuring that no `Element` variable can be created without its limb being properly constrained. This could be done in the compiler `Compile()` function in the `parseCircuit()` logic that walks through the circuit's public and private inputs.

Client Response. Won't fix ("We have considered another approach, but the current approach allows us to decouple non-native arithmetic from the circuit frontend design, keeping it relatively lightweight.")