ZKSECURITY

Audit of Linea Wizard's crypto/ & math/

June 17th, 2024

Introduction

On June 17th, 2024, Linea requested an audit of the wizard crypto and math libraries, as part of their zkEVM project. The audit was conducted by two zkSecurity engineers over a period of three weeks. The audit focused on the correctness and security of parts of the libraries, as well as the safety of their interface.

We found the code to be well documented and organized.

Scope

The scope of the audit included two inner components of the zkEVM repository.

The `` `math/` `` component, which consisted of:

- Definition of the scalar field of the BLS12-377 curve.
- An implementation of FFT functions and related polynomial utility functions.
- An implementation of Smartvectors, an abstraction over field element vectors.

The `` `crypto/` `` component, which consisted of:

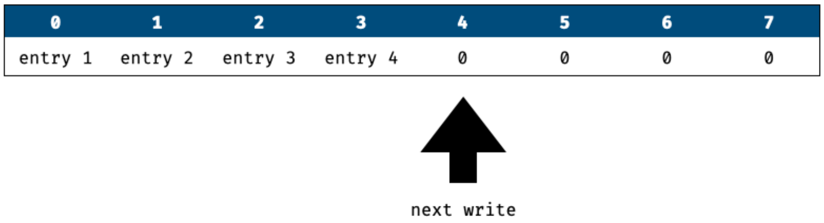
- Three cryptographic hash implementations: Keccak, MiMC, and RingSIS.
- A sparse merkle tree implementation, and an abstraction for a key-value store built as a wrapper around the sparse merkle tree.
- A code-based polynomial commitment scheme: Vortex.

Below, we provide an overview of some of the subcomponents, before diving into findings and recommendations.

The math/ Accumulator and Sparse Merkle Tree

In order to enable verifiable accesses to the state of the world (in Linea), the zkEVM repository implements a key-value store on top of a sparse merkle tree implementation. In this section we describe exactly how.

A doubly-linked list for a key-value store. A continuous array is used to emulate a contiguous region of memory that can be written to in order from left to right, in a write-once fashion.



0	1	2	3	4	5	6	7
entry 1	entry 2	entry 3	entry 4	0	0	0	0

This array of memory is used to store a doubly-linked list. To do that, the first two entries are initialized with the head and the tail of the doubly-linked list. An element of the doubly-linked list is a tuple `(hash(key), hash(value))` and the doubly-linked list is ordered by the hashed key of its elements.

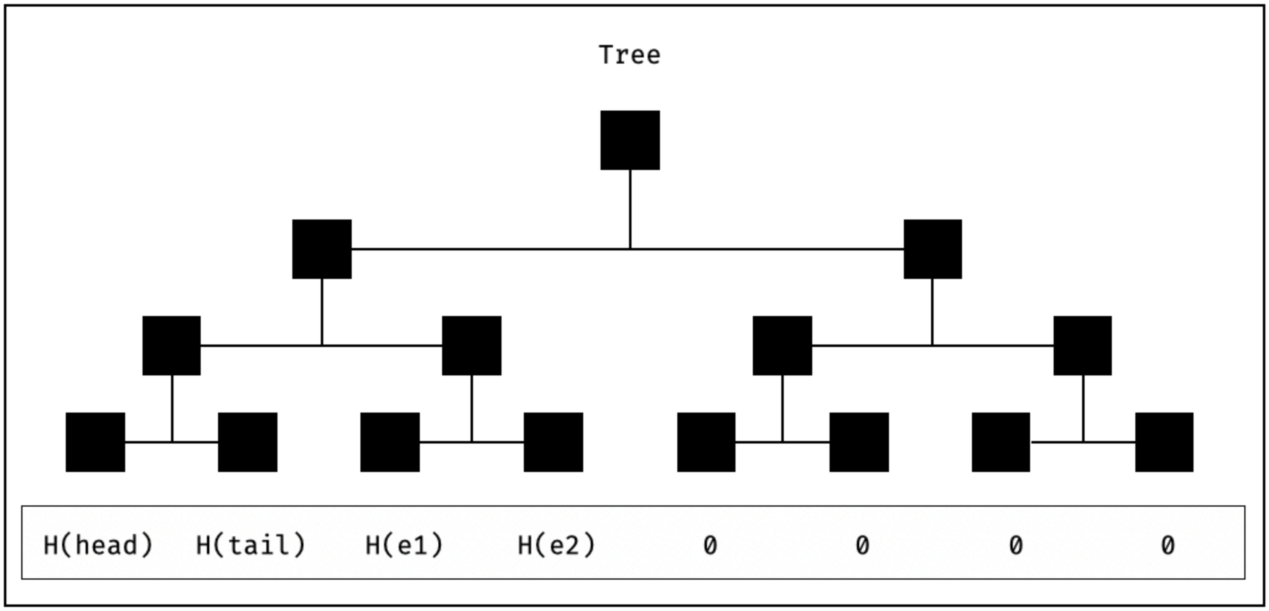
Update to the key-value store. The doubly-linked list serves as a nice way to provide verifiable updates to the key-value store. For example, *insertions* can prove that they are inserting a key in the right place in the doubly-linked list, by proving that:

- the previous element in the list has a smaller hashed key
- the next element in the list has a bigger hashed key
- these "sandwiching elements" points at each other, and can be updated to point at the newly inserted element instead

In addition, the key-value store can make its entries unique by enforcing strict comparisons between the hashed keys.

Merkelizing the memory region. In order to record the state of the world between proved state transitions, and verifiably access and update the doubly-linked-list, the memory containing the elements of the list is merkelized.

This way, its root can be passed in between state transitions, and its state can be deconstructed to verifiably access entries of the heap and update them. We provide an example of such a merkelized memory region (containing a simple doubly-linked list) below.



Data

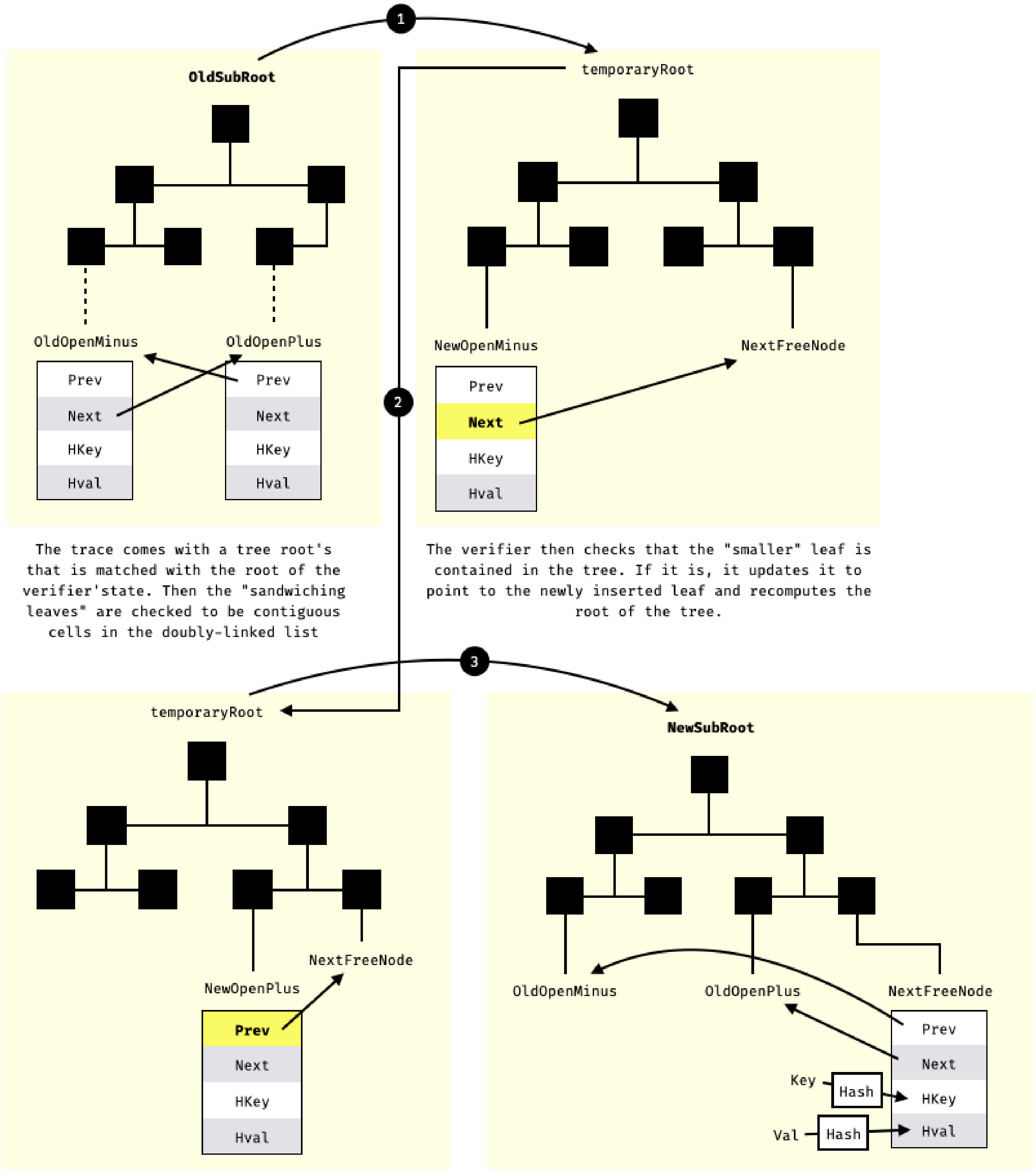
```
0 → head = { prev: 0 (head), next: 2 (e2), hkey: 0, hval: 0 }

1 → tail = { prev: head, next: 1 (tail), hkey: 0, hval: 0 }

2 → e1 = { prev: 3 (e2), next: 1 (tail), hkey: H("foo"), hval: H("bar") }

3 → e2 = { prev: head, next: e1, hkey: H("hello"), hval: H("world") }
```

Implementation details. Note that the implementation accesses and updates the merkle tree in multiple transitions instead of a single atomic update. We illustrate this in the following diagram.



The trace comes with a tree root's that is matched with the root of the verifier'state. Then the "sandwiching leaves" are checked to be contiguous cells in the doubly-linked list

The verifier then checks that the "smaller" leaf is contained in the tree. If it is, it updates it to point to the newly inserted leaf and recomputes the root of the tree.

It then does the same thing with the "bigger" leaf. Notice that the temporary root is used to prove that the leaf is in the tree, and to update that leaf.

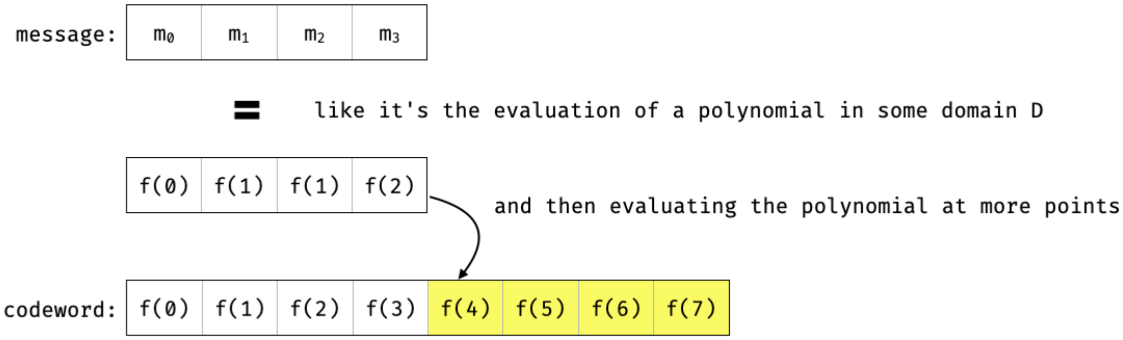
Finally, the new leaf is inserted and checked to correctly link to the "sandwhiching" leaves. The new root is computed (and only if a Merkle proof shows that there was an empty leaf previously at the NextFreeNode path).

The Vortex polynomial commitment scheme

In this section we briefly explain how the Vortex PCS works.

First, Vortex is based on codes. So let's briefly mention how encoding using Reed-Solomon works. Imagine that you want to encode a message $\mathrm{m} = [\mathrm{m}_0, \mathrm{m}_1, \mathrm{m}_2, \mathrm{m}_3]$ into a larger codeword. The idea is that an invalid message is easier to observe if it is stretched into a larger array of values (which would amplify its anomalies).

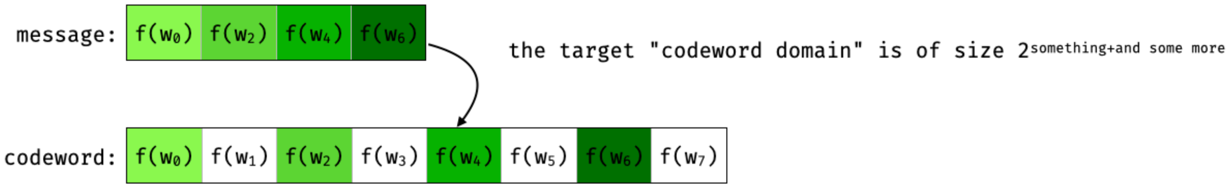
The Reed-Solomon code does that by looking at the message as if it contains the coefficients or the evaluations of a polynomial, and then evaluate that polynomial at *more* points. For example:


message:
m_0
m_1
m_2
m_3
=
like it's the evaluation of a polynomial in some domain D
f(0)
f(1)
f(1)
f(2)
and then evaluating the polynomial at more points
codeword:
f(0)
f(1)
f(2)
f(3)
f(4)
f(5)
f(6)
f(7)

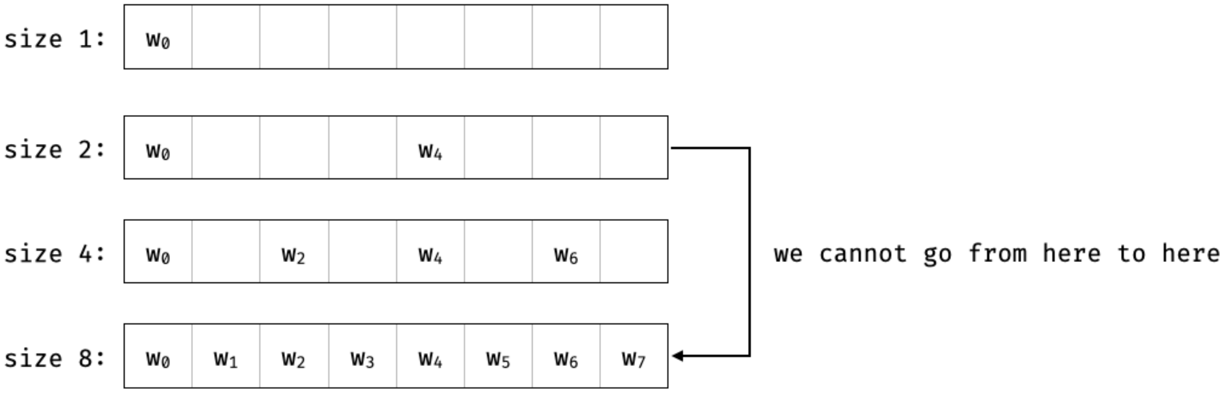
In the previous example we choose to see the message like it is a polynomial in Lagrange form (we have its evaluations over some domain). Since the domain evaluated on is an extension that preserves the points of the original domain, we obtain a *systematic* code: the original message is embedded in the codeword.

In ZK, we deal with arithmetic circuits and so we are interested in a domain that's a multiplicative subgroup in a field (specifically a subgroup of order a power of two, because this plays nice with FFTs, and FFTs allow us to quickly evaluate a polynomial on a domain, while iFFT allows us to quickly interpolate a polynomial from its evaluations).

In our previous example, the original "message domain" was D of size $2^{\text{something}}$. So what happens if we change the domain? The original elements of the message are still embedded in the codeword but at different positions: this is still a systematic code.

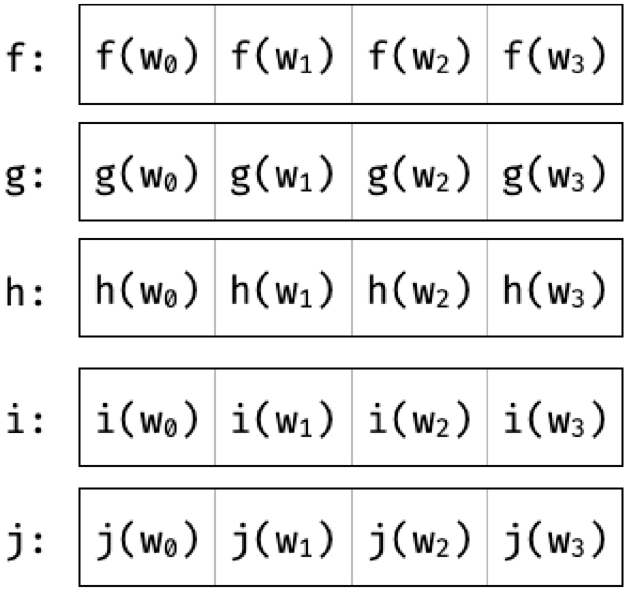


This is because of how we find embedded subgroups of order powers of two in our field:



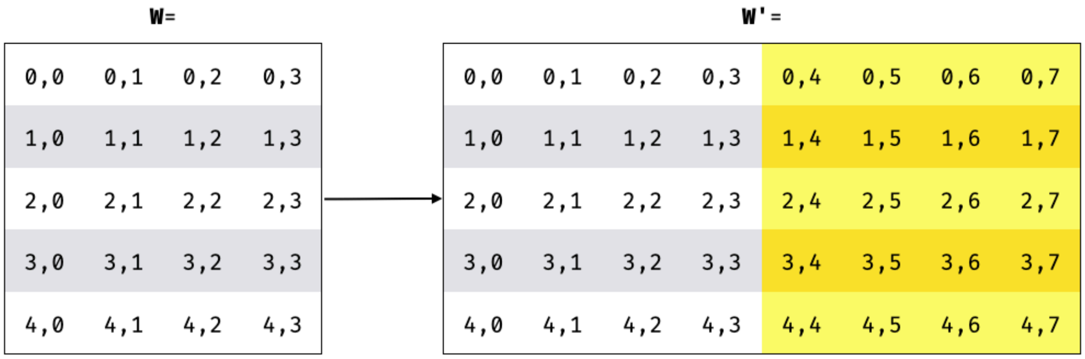
Back to what Vortex provides: it allows you to get the evaluations of a number of polynomials at the same point.

So we start with a number of polynomials in lagrange form (their evaluations over some points $w_0, w_1, w_2, \cdots$) which you can see as a matrix $\mathbf{W}$:



f:	$f(w_0)$	$f(w_1)$	$f(w_2)$	$f(w_3)$
g:	$g(w_0)$	$g(w_1)$	$g(w_2)$	$g(w_3)$
h:	$h(w_0)$	$h(w_1)$	$h(w_2)$	$h(w_3)$
i:	$i(w_0)$	$i(w_1)$	$i(w_2)$	$i(w_3)$
j:	$j(w_0)$	$j(w_1)$	$j(w_2)$	$j(w_3)$

In Vortex, the prover encodes all the rows of the matrix, getting a larger matrix. The greater the "blowup factor" of the Vortex parameters, the more columns we get. And since we have a systematic code, it preserves the original messages on each row:



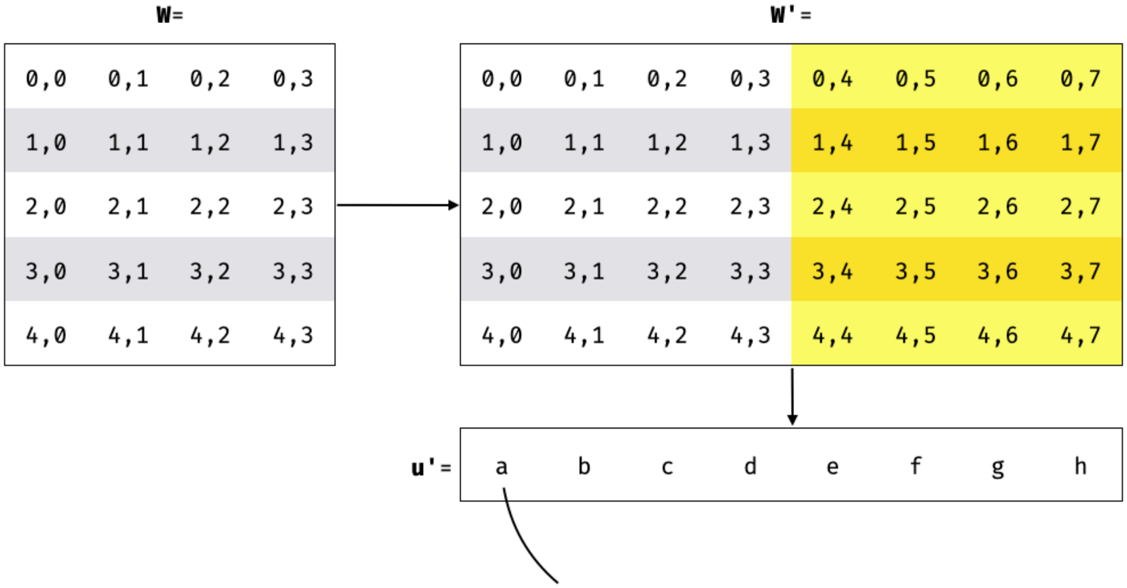
W=

0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3
2,0	2,1	2,2	2,3
3,0	3,1	3,2	3,3
4,0	4,1	4,2	4,3

W'=

0,0	0,1	0,2	0,3	0,4	0,5	0,6	0,7
1,0	1,1	1,2	1,3	1,4	1,5	1,6	1,7
2,0	2,1	2,2	2,3	2,4	2,5	2,6	2,7
3,0	3,1	3,2	3,3	3,4	3,5	3,6	3,7
4,0	4,1	4,2	4,3	4,4	4,5	4,6	4,7

From there, the verifier samples a random challenge β, which the prover uses to take a linear combination of all the encoded rows:



W=

0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3
2,0	2,1	2,2	2,3
3,0	3,1	3,2	3,3
4,0	4,1	4,2	4,3

W'=

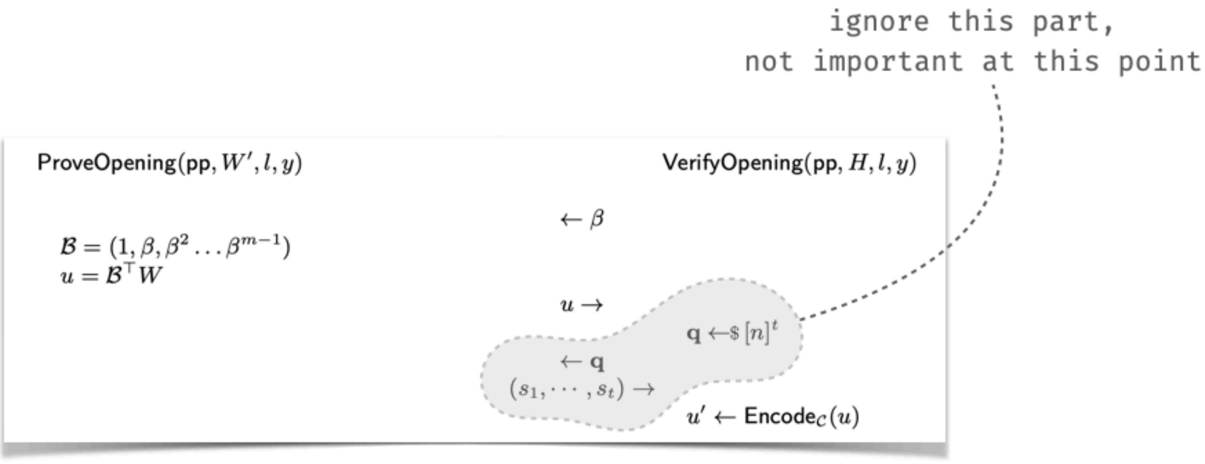
0,0	0,1	0,2	0,3	0,4	0,5	0,6	0,7
1,0	1,1	1,2	1,3	1,4	1,5	1,6	1,7
2,0	2,1	2,2	2,3	2,4	2,5	2,6	2,7
3,0	3,1	3,2	3,3	3,4	3,5	3,6	3,7
4,0	4,1	4,2	4,3	4,4	4,5	4,6	4,7

u'=

a	b	c	d	e	f	g	h

for example, a = **W'**[0,0] + β × **W'**[1,0] + β^2 × **W'**[2,0] + β^3 × **W'**[3,0] + β^4 × **W'**[4,0]

In the paper the non-encoded $\mathbf{u}$ is sent instead, and the verifier encodes it. This is equivalent to interpolating $\mathbf{u}'$ and checking that it's a polynomial of degree at most $d = m - 1$ (which is what the implementation does):



ignore this part,
not important at this point

$\mathsf{ProveOpening}(\mathsf{pp}, W', l, y)$ $\mathsf{VerifyOpening}(\mathsf{pp}, H, l, y)$

$\leftarrow \beta$

$\mathcal{B} = (1, \beta, \beta^2 \dots \beta^{m-1})$
$u = \mathcal{B}^\top W$

$u \rightarrow$

$\mathbf{q} \leftarrow\$ [n]^t$

$\leftarrow \mathbf{q}$

$(s_1, \cdots, s_t) \rightarrow$

$u' \leftarrow \mathsf{Encode}_{\mathcal{C}}(u)$

We know that $\mathbf{u}'$ interpolates to $\mathrm{Int_u(x)}$ the linear combination of all polynomials. i.e. $\mathrm{Int_u(x) = f(x) + \beta \cdot g(x) + \beta^2 \cdot h(x) + \beta^3 \cdot i(x) + \beta^4 \cdot j(x)}$.

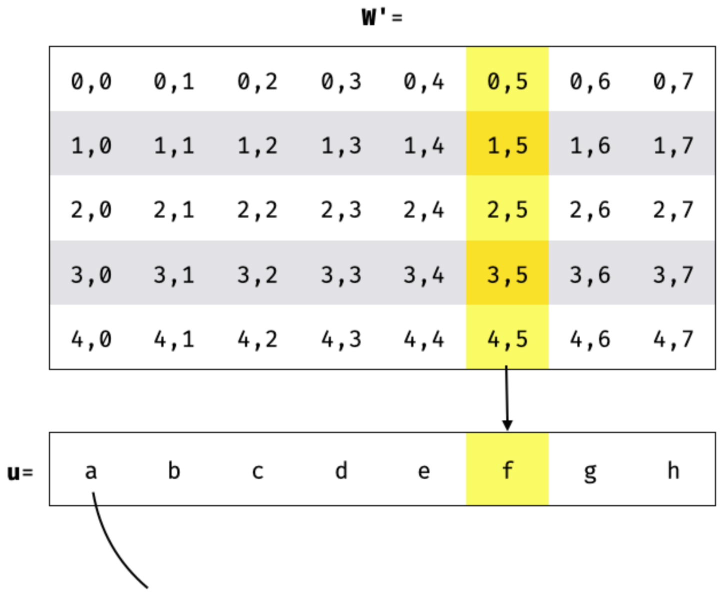
As such, if we want to check that for some z we have

$$\mathrm{f(z) = y_f, g(z) = y_g, h(z) = y_h, i(z) = y_i, j(z) = y_j}$$

we can check that

$$\mathrm{Int_u(z) = y_f + \beta \cdot y_g + g(x) + \beta^2 \cdot y_h + h(x) + \beta^3 \cdot y_i + i(x) + \beta^4 \cdot y_j}$$

The verifier can then query random columns, and check that the linear combination is correctly formed for the matching entry



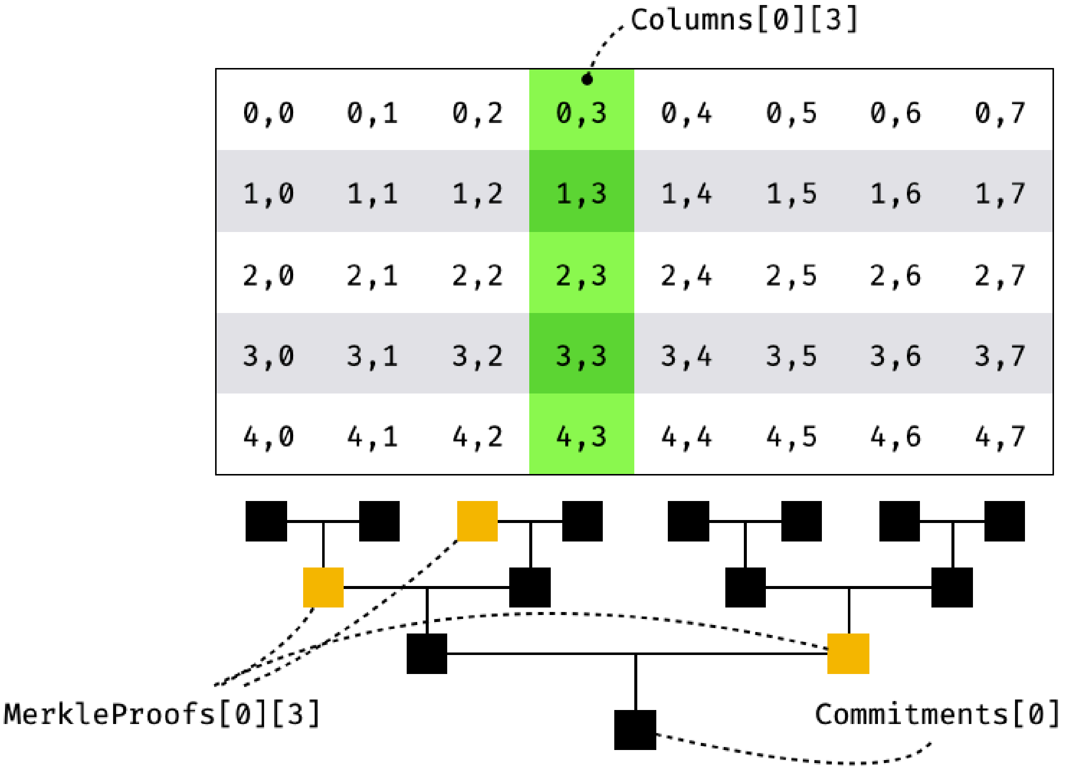
0,0	0,1	0,2	0,3	0,4	0,5	0,6	0,7
1,0	1,1	1,2	1,3	1,4	1,5	1,6	1,7
2,0	2,1	2,2	2,3	2,4	2,5	2,6	2,7
3,0	3,1	3,2	3,3	3,4	3,5	3,6	3,7
4,0	4,1	4,2	4,3	4,4	4,5	4,6	4,7

a	b	c	d	e	f	g	h

by checking if f = **W'**[0,5] + β × **W'**[1,5] + β^2 × **W'**[2,5] + β^3 × **W'**[3,5] + β^4 × **W'**[4,5]

> *"Implementation note: as the verifier needs to dynamically index into the linear combination vector $\mathbf{u}'$, it places the vector values in an indexed lookup table and retrieves its entries by performing table lookups."*

The verifier obviously wants to verifiably query the encoded matrix $\mathbf{W}'$ in an efficient way (without having to know the whole matrix). To do that, the RS-encoded matrix is committed to (using a Merkle tree), and only the columns that are queried are sent to the verifier (accompanied by merkle proofs):



0,0	0,1	0,2	0,3	0,4	0,5	0,6	0,7
1,0	1,1	1,2	1,3	1,4	1,5	1,6	1,7
2,0	2,1	2,2	2,3	2,4	2,5	2,6	2,7
3,0	3,1	3,2	3,3	3,4	3,5	3,6	3,7
4,0	4,1	4,2	4,3	4,4	4,5	4,6	4,7

A note on Ringsis modulo $X^n - 1$

The RingSIS implementation is mostly a wrapper over a core implementation. Although it does contain a few more helper functions, including a function to compute the result of the hash modulo the $X^n - 1$ polynomial.

The function's goal is to get the result of the multiplication $a(x) \cdot b(x) = r(x) \mod x^n - 1$, which is the same as:

$$a(x) \cdot b(x) = r(x) + (x^n - 1) \cdot t(x)$$

for some quotient polynomial $t(x)$

notice that $x^n - 1$ is the vanishing polynomial for the n-th roots of unity, so for any n-th root of unity ω we have:

$$a(\omega) \cdot b(\omega) = r(\omega) + (\omega^n - 1) \cdot t(\omega) = r(\omega) + 0 \cdot t(\omega) = r(\omega)$$

so we can get n evaluations of $r(x)$ by evaluating the left-hand side on all n n-th roots of unity, then interpolate (since r is of degree $n - 1$, we just need n evaluations).

Using SmartVectors for Optimizing Mathematical Operations

We further provide some background on smartvectors, an important abstraction used from optimizing math operations in Linea's implementation.

The Linea prover defines a specialized abstraction over vectors to optimize both memory consumption and performance when working with vectors of field elements. This abstraction is smartvectors, which are implemented by four different types designed to handle specific common vector types. The types of smartvectors are `` `Regular` ``, `` `Constant` ``, `` `Rotated` ``, and `` `PaddedCircularWindow` ``.

Each smartvector type implements specific functions such as retrieving its length, accessing a specific element, producing a subvector, and rotating the vector.

Regular represents a normal vector (i.e., slice) of field elements. This is the default type, and every other smartvector implements a function to convert and create a regular smartvector.

```
type Regular []field.Element
```

Constant is a smartvector obtained by repeating the same value for the specified length. This implementation provides maximum memory optimization and speeds up some operations.

```
type Constant struct {
    val    field.Element
    length int
}
```

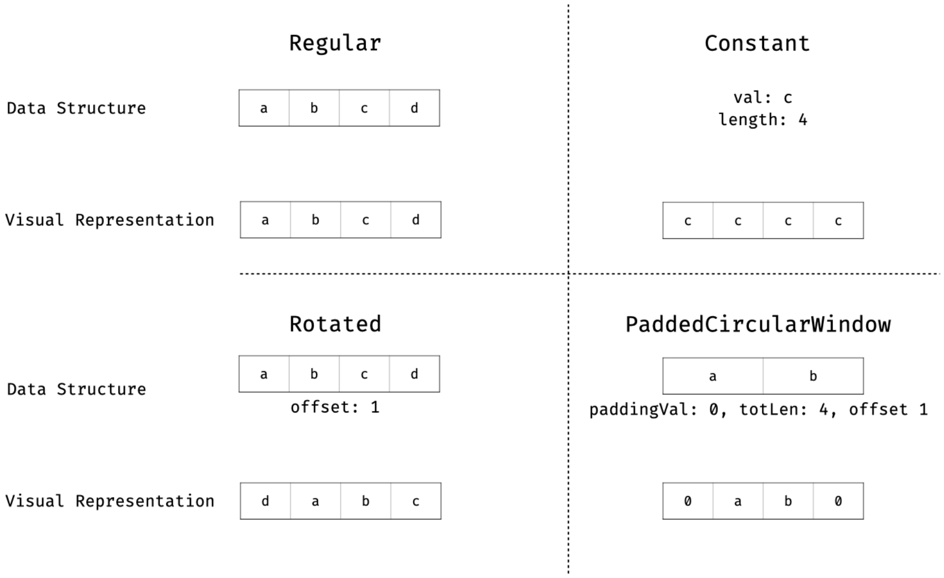
Rotated works by abstractly storing an offset and only applying the rotation when the vector is written, sub-vectored, or evaluated for specific operations. This makes rotations essentially free.

```
type Rotated struct {
    v      Regular
    offset int
}
```

PaddedCircularWindow represents a vector padded up to a certain length and possibly rotated. It combines optimizations from all previous smartvectors and implements extensive optimizations when performing arithmetic operations (c.f., `` `processWindowedOnly` `` in `` `windowed.go` ``).

```
type PaddedCircularWindow struct {
    window     []field.Element
    paddingVal field.Element
    totLen, offset int
}
```

The following figure visualizes the different types of smartvectors.



Usage and Benefits

Smartvectors are used for memory and performance optimizations. They play a crucial role in performing FFTs on the Linea prover, representing polynomials, and performing optimized operations such as batch interpolation by reducing computations through pre-processing constant vectors in advance.

The main operations involving smartvectors are defined in **`arithmetic_basic.go`**, **`arithmetic_gen.go`**, **`arithmetic_op.go`**, **`fft.go`**, and **`polynomial.go`**. A critical function is **`processOperator`** in **`arithmetic_gen.go`**, which handles various operations and performs optimizations based on the type of input smartvectors.

Precautions and Testing

While smartvectors are designed for optimization purposes, they should be used with care. The API is designed to prevent mutating smartvectors directly; instead, users should produce new instances. During our audit, we looked for potential issues arising from API misuse, identifying corner cases that could cause unexpected behavior, and ensured that the API was clear and not prone to misuse. We also ran multiple test cases and tweaked the existing test suite to validate that smartvectors work as expected.

In summary, despite the complexity of some optimizations, the code is well-documented and thoroughly tested. During our audit, we identified the following issues:

- "Missing Check and Integer Overflow In Rotated Smartvectors Offset Handling"
- "Loose Bound Checks In Subvector For Windowed Smartvectors"
- "Zero and Negative Length Smartvectors"
- "Rotated Smartvector's API is confusing"

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	utils	Incorrect Check For Power Of Two	Low
01	utils	`NextPowerOfTwo` Does Not Work With All Values	Low
02	maths/common/smartvectors	Missing Check and Integer Overflow In Rotated Smartvectors Offset Handling	Low
03	crypto/state-management/smt	Sparse Merkle Tree Check Is Too Loose	Low
04	crypto/state-management/smt	Incorrect Padding For Sparse Merkle Tree Internal Nodes	Low
05	crypto/state-management/accumulator	Unconstrained `NextFreeNode` Values In Accumulator Traces	Low
06	maths/common/smartvectors	Rotated Smartvector's API is confusing	Informational

ID	COMPONENT	NAME	RISK
07	maths/common/smartvectors	Loose Bound Checks In Subvector For Windowed Smartvectors	Informational
08	maths/common/smartvectors	Zero and Negative Length Smartvectors	Informational

# 00 - Incorrect Check For Power Of Two

● utils

Low

Description. In the codebase there are many calls to the function `` `IsPowerOfTwo` ``, which returns true if the input provided is a power of two. This check is accomplished by doing a bitwise manipulation of the input: a power of two can be represented in binary as having only one `` `1` `` in its representation. We can therefore check if a number is a power of two by computing the bitwise and between the input and its predecessor, excluding zero.

```go
// Return true if n is a power of two
func IsPowerOfTwo[T ~int](n T) bool {
    return n&(n-1) == 0 && n != 0
}
```

Unfortunately, this check is correct only for *unsigned* integers, and for signed integers it allows one more negative value, which is -2^{63} (assuming 64-bit signed integers). The signed binary representation of this value is indeed `` `1` `` in the most significant bit, and all `` `0` `` in the other places.

The input was actually found using the z3 SMT solver, imposing the constraints that the test passes and the value to be negative. Interestingly, this is also the *only* negative value for which this function returns `` `true` ``. Below is a z3 Python script that finds such value.

```python
import z3

x = z3.BitVec('x', 64)
s = z3.Solver()

# add the contraints to pass the check
s.add(x & (x-1) == 0)
s.add(x != 0)

# find a negative value that satisfies the constraints
s.add(x < 0)
s.check()
neg_solution = s.model()[x]
assert neg_solution == -9223372036854775808

# this is the only negative value that passes the checks
s.add(x != -9223372036854775808)
assert s.check() == z3.unsat
```

Indeed, if we try to pass this value to the function in the go implementation, the return value is `` `true` ``.

```
func TestIsPowerOfTwo(* testing.T) {
    fmt.Println(IsPowerOfTwo(-9223372036854775808)) // true
}
```

Recommendation. Modify the check so that the input is strictly positive.

```
-return n&(n-1) == 0 && n != 0
+return n&(n-1) == 0 && n > 0
```

# 01 - `NextPowerOfTwo` Does Not Work With All Values

● utils

Low

Description. The `` `NextPowerOfTwo` `` function computes $\lceil \log_2(v) \rceil$ for a given integer v. It does so by creating a mask of bits all set to 1 and of the same bit-length of v. Then it increments the mask by one to return the next power of two:

```go
func NextPowerOfTwo[T ~int](in T) T {
    v := in
    v--
    v |= v >> (1 << 0)
    v |= v >> (1 << 1)
    v |= v >> (1 << 2)
    v |= v >> (1 << 3)
    v |= v >> (1 << 4)
    v |= v >> (1 << 5)
    v++
    return v
}
```

However, it does not work well with all values. As the function is written to be generic for types `` `int` `` and generally all types whose underlying type is `` `int` ``, and as `` `int` `` is signed by default, some results might not be as expected. For example, when we try to compute the next power of two for $2^{62} + 1$, the function returns a negative result:

```go
func TestNextPowerOfTwo(t *testing.T) {
    // prints `4611686018427387905 -9223372036854775808`
    fmt.Println((1<<62)+1, utils.NextPowerOfTwo((1<<62)+1))
}
```

Recommendation. Either assert that the argument passed is not too large, or constrain the type parameter to be the set of all unsigned integers (`` `~uint` ``). Alternatively, as the function is most likely not meant to be used on such large values, document the assumptions made by the function.

# 02 - Missing Check and Integer Overflow In Rotated Smartvectors Offset Handling

● maths/common/smartvectors

Low

Description. The Rotated Smartvector represents a vector to which is applied a "lazy" rotation. This is accomplished by storing an `offset` variable to represent the rotation offset applied.

The `NewRotated` function aims to enforce some checks, i.e., checking that the vector is not empty and the offset is not larger than the length of the vector.

```go
func NewRotated(reg Regular, offset int) *Rotated {

    // empty vector
    if len(reg) == 0 {
        utils.Panic("got an empty vector")
    }

    // offset larger than the vector itself
    if offset > len(reg) {
        utils.Panic("len %v, offset %v", len(reg), offset)
    }

    return &Rotated{
        v: reg, offset: offset,
    }
}
```

The second `if` statement that checks if the offset is larger than the vector is not complete, as the offset could be a negative value, but its absolute value can still be larger than the length.

Additionally, in the `RotateRight` function, the new `offset` is just added to the stored one without any check, so the resulting offset could be larger than the length.

```go
// Rotates the vector into a new one
func (r *Rotated) RotateRight(offset int) SmartVector {
    return &Rotated{
        v:      vector.DeepCopy(r.v),
        offset: r.offset + offset,
    }
}
```

Further, the offset is implemented using the `int` type, which is susceptible to integer overflow. Since the wrap around of `int` on 64 bit systems is 2^{64}, this could lead to an incorrect operation being applied. Indeed, when we perform the `RotateRight` operation, the offsets are first wrapped by the machine arithmetic, but the rotation should be performed conceptually over the integers modulo the length.

In the example below, we are rotating by a fixed amount, which is zero modulo the length, so the effect of applying the rotation should be none. However, due to the integer wrap around, after five rotations the resulting vector ends up rotated by 4.

```go
func TestRotatedOverflow(_ *testing.T) {
    v := []field.Element{field.NewFromString("1"),
        field.NewFromString("2"),
        field.NewFromString("3"),
        field.NewFromString("4"),
        field.NewFromString("5")};
    r := NewRotated(*NewRegular(v), 0);

    // this should have no effect on the rotated vector, since m % len == 0
    m := 2305843009213693950;
    assertHasLength(m % 5, 0);

    for i := 0; i < 5; i++ {
        r = r.RotateRight(m).(*Rotated);
        fmt.Println(rotatedAsRegular(r).Pretty());
    }

    // Regular[[1, 2, 3, 4, 5]]
    // Regular[[1, 2, 3, 4, 5]]
    // Regular[[1, 2, 3, 4, 5]]
    // Regular[[1, 2, 3, 4, 5]]
    // Regular[[5, 1, 2, 3, 4]] <--- vector rotated by 4
}
```

A similar issue is also present in the Windowed Smartvectors when handling the rotation offset.

Recommendation. Firstly, the check for the length should always be applied and should be updated to check the absolute value of the resulting offset against the length of the smartvector. Additionally, it is recommended to check explicitly for integer overflow every time an operation over machine arithmetics occurs.

# 03 - Sparse Merkle Tree Check Is Too Loose

● **crypto/state-management/smt**

Low

Description. In the sparse merkle tree logic, the `reserveLevel` function is a helper to extend the nodes (potentially leaves) at a certain depth of the tree. The function is meant to be called during new insertions in the tree, if empty nodes (or empty leaves) are impacted in the tree.

The function logic checks that the extension of a level is valid according to the configured-size of the tree:

```go
func (t *Tree) reserveLevel(level, newSize int) {

    // Edge-case, level out of bound
    if level > t.Config.Depth {
        utils.Panic("level out of bound %v", level)
    }

    // Edge : case root level
    if level == t.Config.Depth {
        utils.Panic("tried extending the root level %v", newSize)
    }

    // Will work for both the leaves and the intermediate nodes
    if newSize > 1<<t.Config.Depth-level {
        utils.Panic("overextending the tree %v, %v", level, newSize)
    }
```

In the previous code, the last check is ill-formed, due to the operator precedence of Golang that privileges `<<` above `-`. As such, it is possible to extend a level above its intended shape, potentially allowing undefined behavior.

Note that we could not find any issues in which this would arise, we still recommend fixing the issue.

Recommendation. Fix the previous check by adding parentheses to enforce the order of operations:

```diff
-   if newSize > 1<<t.Config.Depth-level {
+   if newSize > 1<<(t.Config.Depth-level) {
```

In general, always use parentheses in chained operations to avoid relying on remembering the correct order of operator precedence in Golang (and potentially getting it wrong).

# 04 - Incorrect Padding For Sparse Merkle Tree Internal Nodes

● **crypto/state-management/smt**

Low

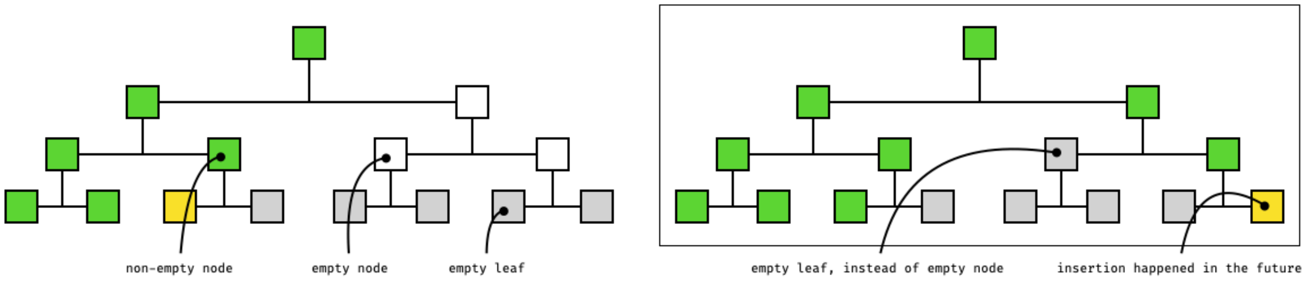
Description. In the sparse merkle tree logic, the `` `reserveLevel` `` helper function can be called during insertions to extend the tree's nodes at specified depths. For new leaves, empty leaves are added as `` `EmptyLeaf()` `` as the zero vector. Interestingly, for new nodes, an empty leaf is also used:

```
func (t *Tree) reserveLevel(level, newSize int) {
    // TRUNCATED...

    if newSize <= len(t.OccupiedNodes[level-1]) {
        // already big enough
        return
    }
    // else, we add extra "empty leaves" at the end of the slice
    padding := make([]types.Bytes32, newSize-len(t.OccupiedNodes[level-1]))
    for i := range padding {
        padding[i] = EmptyLeaf()
    }
    t.OccupiedNodes[level-1] = append(t.OccupiedNodes[level-1], padding...)
```

The code essentially assumes that this is benign, because it will always be called in insertions that incrementally go from the most-left side of the tree to the most-right side. In which case, new nodes at any level are always added in order, one at a time, and immediately get updated.

As far as we could tell, this is how the current logic uses the sparse merkle tree, which leads to no issue. But a sparse-merkle tree does not necessarily have to be used this way, for example it could decide to insert leaves at positions "in the future", leaving gaps in the leaves, in which case the previous code would break the sparse merkle tree. We illustrate this example in the following diagram:


non-empty node
empty node
empty leaf
empty leaf, instead of empty node
insertion happened in the future

Recommendation. Fix the code above to use the already-available precomputed empty nodes of the right level instead of empty leaves:

```diff
        padding := make([]types.Bytes32, newSize-len(t.OccupiedLeaves))
        for i := range padding {
-           padding[i] = EmptyLeaf()
+           padding[i] = t.EmptyNode[level-1]
        }
```

# 05 - Unconstrained `NextFreeNode` Values In Accumulator Traces

- crypto/state-management/accumulator

Low

Description. In the accumulator implementation, the prover emits a **trace** object for every operation that is performed. The verifier then can then check the validity of such traces, and consequently the validity of the operation. The traces contain all the information that is needed to the verifier to audit the correctness of the operation, such as leaf openings, their respective Merkle proofs, Merkle tree roots and so on.

Different operations have different traces fields, but every one contains the value of the `` `NextFreeNode` `` of the accumulator. This field is called `` `NextFreeNode` `` in read zero and read non-zero operations, and `` `NewNextFreeNode` `` in insertion, deletion and update operations. Those values in the traces are never read by the verifier functions, so they remain completely unconstrained. This entails that it is possible to change those values freely, and the verifier will still accept the trace as valid. We note that this behaviour is present in all the operations.

Below is a test case showing that changing the `` `NewNextFreeNode` `` in an insertion trace still passes the verification checks.

```go
func TestNextFreeNode(t *testing.T) {
    acc := newTestAccumulatorKeccak()
    ver := acc.VerifierState()

    // Insert a value
    trace := acc.InsertAndProve(dumkey(0), dumval(0))

    // Change the NewNextFreeNode in the trace
    // to an arbitrary value
    trace.NewNextFreeNode = -42

    // The verification still passes
    err := ver.VerifyInsertion(trace)
    require.NoErrorf(t, err, "verification")
}
```

Note that the verifier state keeps its own `` `NextFreeNode` `` value internally, so those values in the traces are fundamentally not needed. Even if somehow we could tamper the verifier state, e.g., by making it change its `` `NextFreeNode` `` internal value to an arbitrary value, we could not find any issues undermining the consistency of the underlying key-value mapping. Indeed, the inner Sparse Merkle Tree would still retain its own consistency thanks to the Merkle proofs verifications currently in place. What could happen in this scenario, however, is that valid operations may end up being rejected. For example the insertion operation expects the position of the new node to be equal to `` `NextFreeNode` ``, and that the overwritten node was initially empty.

Recommendation. Implement checks in all the trace verification functions for consistency of the `` `NextFreeNode` `` and `` `NewNextFreeNode` `` values between the traces and the one kept by the verifier state. Even if we could not find any accumulator consistency issues, we still recommend to apply this remediation to ensure the internal consistency of the traces, and for avoiding potential issues coming from future uses of the trace fields.

# 06 - Rotated Smartvector's API is confusing

- **maths/common/smartvectors**

Informational

Description. The Rotated Smartvector is used to optimize rotated vectors of field elements by applying lazy rotations. To accomplish that, the vector is stored in a variable, and the other variable takes track of the offset.

The `RotateRight` is a function that should rotate the vector by a given `offset`.

```
func (r *Rotated) RotateRight(offset int) SmartVector {
    return &Rotated{
        v:      vector.DeepCopy(r.v),
        offset: r.offset + offset,
    }
}
```

Despite its name suggesting that the vector will be rotating to the right, the vector is rotated to the left. For example, if we have the following vector:

```
[0 1 2 3 4]
```

And we rotate it right two positions, we would expect the result to be:

```
[3 4 0 1 2]
```

But it is

```
[2 3 4 0 1]
```

Although this API is functionally correct (if the expected behavior is to rotate left), it can be confusing and introduce subtle errors. `NewRotated` also rotates to the left and should be clarified in the documentation if that is the expected behavior.

Recommendation. It is recommended to either change the logic of the `RotateRight` function or to clarify the expected behavior in the documentation of the function.

# 07 - Loose Bound Checks In Subvector For Windowed Smartvectors

- **maths/common/smartvectors**

Informational

Description. In the windowed smartvector code, the `SubVector` function is intended to be used to extract a sub-vector with given start and stop indices. The function performs bound checks for the input parameters `start` and `stop`, however those checks do not account for the possibility of `start` to be negative.

```
func (p *PaddedCircularWindow) SubVector(start, stop int) SmartVector {
    // Sanity checks for all subvectors
    assertCorrectBound(start, p.totLen)
    // The +1 is because we accept if "stop = length"
    assertCorrectBound(stop, p.totLen+1)

    if start > stop {
        panic("rollover are forbidden")
    }
```

Later in the function, the length of the target sub-vector `b` is computed as `stop - start`.

```
    b := stop - start
    c := normalize(p.interval().start(), start, n)
    d := normalize(p.interval().stop(), start, n)
```

In the example below we are constructing a windowed smartvector and taking the sub-vector of bounds `-200` and `5`. The resulting sub-vector length `b` will be computed as `stop - start`, and it will result in the value `205`, thus it will be larger than the original one (notice the `totlen` fields in the outputs).

```
func TestWindowed(_ *testing.T) {
    v := []field.Element{field.NewFromString("1"),
        field.NewFromString("2"),
        field.NewFromString("3"),
        field.NewFromString("4"),
        field.NewFromString("5")};
    r := NewPaddedCircularWindow(v, field.Zero(), 1, 16);

    fmt.Println(r.Pretty());
    // Windowed[totlen=16 offset=1, paddingVal=0, window=[1, 2, 3, 4, 5]]

    w := r.SubVector(-200, 5);
    fmt.Println(w.Pretty());
    // Windowed[totlen=205 offset=9, paddingVal=0, window=[1, 2, 3, 4, 5]]
}
```

Recommendation. Do not allow negative `` `start` `` values by panicking if `` `start < 0` ``.

# 08 - Zero and Negative Length Smartvectors

- maths/common/smartvectors

Informational

Description. Constructing zero- or negative-length smartvectors should be illegal, but we can construct them using only API functions. Note that for negative-length smartvectors, when you try to convert them to gnark frontend variables, the code will panic. It will create an empty slice of field elements for zero length, which might lead to other more subtle vulnerabilities.

```
func TestZero(_ *testing.T) {
    a := NewPaddedCircularWindow(
        vector.Rand(5),
        field.Zero(),
        0,
        16,
    ).SubVector(0, 0);
    b := NewRegular([]field.Element{field.Zero()}).SubVector(0, 0);
    c := NewConstant(field.Zero(), 0);
    d := NewConstant(field.Zero(), -1);
    e := NewConstant(field.Zero(), 10).SubVector(3, 1);

    assertHasLength(a.Len(), 0);
    assertHasLength(b.Len(), 0);
    assertHasLength(c.Len(), 0);
    assertHasLength(d.Len(), -1);
    assertHasLength(e.Len(), -2)
}
```

A similar issue might arise when constructing a new `circularInterval`, where there is an implicit assumption that empty intervals should never be constructed. Although the `ivalWithStartLen` function includes a check to prevent empty intervals, the `ivalWithFullLen` function does not perform any such validation. This inconsistency could potentially lead to unexpected behavior or errors if empty intervals are inadvertently created using `ivalWithFullLen`. It is crucial to ensure that all interval construction functions enforce the same validation rules to maintain consistency and prevent potential issues.

Recommendation. To solve the issue, the functions should panic whenever we try to construct an ill-formed smartvector.