Curta / Plonky2x Audit Report

Audited by Allen Roh (rkm0959). Report Published by KALOS.

ABOUT US

Pioneering a safer Web3 space since 2018, KALOS proudly won 2nd place in the Paradigm CTF 2023 with external collaborators. As a leader in the global blockchain industry, we unite the finest in Web3 security expertise. Our team consists of top security researchers with expertise in blockchain/smart contracts and experience in bounty hunting. Specializing in the audit of mainnets, DeFi protocols, bridges, and the ZKP circuits, KALOS has successfully safeguarded billions in crypto assets.

Supported by grants from the Ethereum Foundation and the Community Fund, we are dedicated to innovating and enhancing Web3 security, ensuring that our clients' digital assets are securely protected in the highly volatile and ever-evolving Web3 landscape.

Inquiries: audit@kalos.xyz
Website: https://kalos.xyz (https://kalos.xyz)

Executive Summary

Purpose of this Report

This report was prepared to audit the overall security of the proof system implementation developed by the Succinct team. KALOS conducted the audit focusing on the ZKP circuits as well as the prover and verifier implementations. In detail, we have focused on the following.

- Completeness of the ZKP Circuits

- Soundness of the ZKP Circuits
- Correct Implementation of Fiat-Shamir Heuristic
- Sound usage of the recently developed cryptographic techniques
  - Lookups via Logarithmic Derivative
  - Offline Memory Checking Techniques

Codebase Submitted for the Audit

https://github.com/succinctlabs/curta (https://github.com/succinctlabs/curta)

- commit hash `11f52ddc96a2433d1d983668c484a54afefc1f18`
  - `air/extension/cubic.rs`
  - `plonky2/stark/verifier.rs`
  - `plonky2/cubic`
  - `chip/constraint/mod.rs`
  - `chip/instruction`
  - `chip/field`
  - `chip/ec`
  - `chip/uint`
  - `chip/memory`
  - `chip/table`
  - `machine/bytes/stark.rs`
  - `machine/emulated/stark.rs`
  - `machine/ec/builder.rs`
  - `machine/hash/blake`
    - commit hash `59e489d21622a92b25a2e4a5b710ce322f309b87`
  - `machine/hash/sha/algorithm.rs`
  - `machine/hash/sha/sha256/air.rs`
  - `machine/hash/sha/sha512/air.rs`

https://github.com/succinctlabs/succinctx/tree/main/plonky2x/core/src/frontend (https://github.com/succinctlabs/succinctx/tree/main/plonky2x/core/src/frontend)

- commit hash `dcf82c8da1be4d7a84c9930e29f27143124899a9`
  - `builder/mod.rs`
  - `builder/io.rs`
  - `builder/proof.rs`
  - `curta/field/variable.rs`
  - `curta/ec/point.rs`
  - `hash`

- `ecc`
- `uint`
- `merkle`
- `mapreduce`
- `vars`
- `ops/math.rs`

Audit Timeline

- 2023/11/15 ~ 2023/12/19, 5 engineer weeks
- 2024/1/22 ~ 2024/1/26, 1 engineer week

Codebase Overview - Curta

Overall Structure

The overall format of the proof system is defined in the `AirParser` trait, which gives access to the `local` row and the `next` row of the matrix of variables, as well as the `challenges` for the randomized AIR, and the `public slice` and `global slice` for the public/global inputs. It also allows one to set various types of constraints, such as boundary constraints (first/last row) or transition constraints. It also provides an API to perform arithmetic over the variable type and the field type that the current AIR is defined on. Refer to `air/parser.rs` .

The `AirConstraint` trait has the `eval` function, which evaluates the polynomials that should vanish, corresponding to the constraints that we desire to set. By taking the mutable `AirParser` as an input, it calls the `AirParser` to add various constraints. The `RAir` trait also supports `eval_global` function, which corresponds to global constraints.

The columns are often handled using the `Register` trait defined in the `register` folder. Each register has its own `CellType` which is either `bit` or `u16` or just `element` , and its own corresponding `MemorySlice` . `MemorySlice` corresponds to one of `Local` , `Next` , `Public` , `Global` , or `Challenge` , where each of them has the same meaning as the relevant API at the `AirParser` trait. There is also a `ArrayRegister` to handle multiple registers of the same type.

Cubic Extensions

The main field the proof system works on is the Goldilocks field $\mathbb{F}_p$ with $p = 2^{64} - 2^{32} + 1$.

However, for some situations the system works on the extension field

$$\mathbb{F}_{p^3} = \mathbb{F}_p[u]/(u^3 - u + 1)$$

in the case a larger field is needed. One can perform multiplication as

$$(x_0 + x_1u + x_2u^2)(y_0 + y_1u + y_2u^2) =$$
$$(x_0y_0) + (x_0y_1 + x_1y_0)u + (x_0y_2 + x_1y_1 + x_2y_0)u^2 + (x_1y_2 + x_2y_1)(u-1) + (x_2y_2)(u^2 - u) =$$
$$(x_0y_0 - x_1y_2 - x_2y_1) + (x_0y_1 + x_1y_0 + x_1y_2 + x_2y_1 - x_2y_2)u + (x_0y_2 + x_1y_1 + x_2y_0 + x_2y_2)u^2$$

This computation is done (inside the circuit) in files like

- `air/extension/cubic.rs`
- `plonky2/cubic/operations.rs`

Instructions

`instruction/set.rs` defines `AirInstruction` as a enumeration of various types of instructions. `CustomInstruction` is a custom instruction, and `MemoryInstruction` is a memory related instruction that is described later. Here, we discuss other instruction types.

BitConstraint

Simply constrains a `MemorySlice` to be a bit by constraining $x \cdot (x-1) = 0$.

AssignInstruction

Constrains a target `MemorySlice` to be equal to a given source `ArithmeticExpression` , based on the `AssignType` which is either `First` , `Last` , `Transition` , or `All` .

SelectInstruction

Defined in `chip/bool.rs` , given a `BitRegister` `bit` and two `MemorySlice` values `true_value` and `false_value` , outputs a `MemorySlice` `result` computed and constrained as `result = bit * true_value + (1 - bit) * false_value` .

Cycle

Allocates `BitRegister` s `start_bit` and `end_bit` , as well as `ElementRegister` s `element, start_bit_witness, end_bit_witness` . The element constraints are

- `element = 1` on the first row
- `element_next = g * element` as a transition constraint

which shows that `element` are the consecutive powers of g.

Here, g is a generator of a subgroup of $\mathbb{F}_p^\times$ which has the size 2^k.

The `start_bit` constraints are

- `start_bit * (element - 1) == 0` on all rows
- `start_bit_witness * (element + start_bit - 1) == 1` on all rows

The first constraint shows that `start_bit = 1` forces `element = 1`. The second constraint shows that if `element = 1`, then `start_bit` must be equal to `1` as otherwise `element + start_bit - 1 = 0`. It's also clear that when `element != 1` and `start_bit = 0`, it's possible to find a corresponding `start_bit_witness`. This shows that these constraints satisfy completeness and soundness to enforce `start_bit == 1 <=> element = 1`.

The `end_bit` constraints are similar,

- `end_bit * (element - g^-1) == 0` on all rows
- `end_bit_witness * (element + end_bit - g^-1) == 1` on all rows.

A similar logic shows `end_bit == 1 <=> element = g^-1`.

ProcessIdInstruction

Simply constrains that `process_id == 0` on the first row, and `process_id_next = process_id + end_bit` holds as a transition constraint.

Filtered Instruction

It takes any other instruction, and multiplies an `ArithmeticExpression<F>` before constraining it to be zero. One can understand that this is a "filtered" instruction, as in the case where this multiplier evaluates to zero, the original constraint is practically not applied. This functionality is implemented via the `MulParser` struct, which multiplies a `multiplier` before constraining.

Field

The `field` folder deals with modular arithmetic over a fixed prime q. Each element is represented with l limbs where each limbs are represented with a `U16Register`, range checked to be within `u16` range. Here, q is assumed to have no more than $16l$ bits. By considering the limbs as a polynomial, one can consider the full value of the l limbs as a polynomial evaluation at $x = 2^{16}$. This is the main trick to constrain the arithmetic in $\bmod{q}$.

For example, consider that we are dealing with a 256-bit prime q and want to show

$$a + b \equiv res \pmod{q}$$

By considering the 16 limbs as a coefficient to a degree 15 polynomial, it suffices to show

$$a(x) + b(x) - res(x) - carry(x) \cdot q(x) = 0$$

at $x = 2^{16}$. This is equivalent to

$$a(x) + b(x) - res(x) - carry(x) \cdot q(x) = (x - 2^{16}) \cdot w(x)$$

for some polynomial $w(x)$. This pattern appears for other types of arithmetic -

- Subtraction: $a - b = c$ is equivalent to $b + c = a$, so addition can be used.
- Multiplication: The left hand side is changed to $a(x) \cdot b(x) - res(x) - carry(x) \cdot q(x)$
- Division: We check $b \cdot inv = 1$ and $a \cdot inv = res$ using the multiplication gadget.
- "Den": This is a special instruction used in the edward curve computation.
  - If `sign = 1`, then we wish to compute $a / (b + 1)$. This can be done with $(b(x) + 1) \cdot res(x) - a(x) - carry(x) \cdot q(x)$ as the left hand side.
  - If `sign = 0`, then we wish to compute $a / (p - b + 1)$. This can be done with $(b(x) - 1) \cdot res(x) + a(x) - carry(x) \cdot q(x)$ as the left hand side.

All the `AirConstraint`'s `eval` function work in the following way - it first computes the left hand side - the polynomial that should vanish at $x = 2^{16}$, with the parser's API. A `PolynomialParser` is implemented in `polynomial/parser.rs` to aid implementation. Then, it constrains that the polynomial is equal to $(x - 2^{16}) w(x)$ in the `field/util.rs`'s `eval_field_operation` function. This function works as follows.

The function takes in two `U16Register` arrays of size $2l - 2$ called w_{low} and w_{high}. These are regarded as degree $2l-3$ polynomials. These two polynomials are intended to satisfy

$$w_{low}(x) + 2^{16} \cdot w_{high}(x) = w(x) + \text{WITNESS\_OFFSET} \cdot (1 + x + \cdots + x^{2l-3})$$

where `WITNESS_OFFSET` is defined in the `FieldParameters`. Correctly setting `WITNESS_OFFSET` is critical for both completeness and soundness of the field arithmetic constraint system. For a related bug, refer to vulnerability #1.

We note that `res` does not necessarily have to be fully reduced modulo q.

Completeness

The first part for completeness is showing that $carry$ can be computed accordingly. This can be easily shown by proving that $0 \le carry < 2^{16l}$ holds - if all inputs a, b are all reduced modulo q and encoded as the fully reduced values, this can be shown by simple bounding.

The second part, and the more difficult part for completeness is the existence of w_{low} and w_{high}. It suffices to show that all coefficients of the $w(x)$ are within the range

$$[-\text{WITNESS\_OFFSET}, 2^{32} - \text{WITNESS\_OFFSET})$$

Consider the equation

$$a_0 + a_1 x + \cdots + a_n x^n = (x - 2^{16}) (b_0 + b_1 x + \cdots + b_{n-1} x^{n-1})$$

In this case, we have

$$|b_0| = |a_0| / 2^{16}, \quad |b_i| = |b_{i-1} - a_i| / 2^{16} \le (|b_{i-1}| + |a_i|) / 2^{16}$$

Therefore, one can bound $|b_i|$ by bounding $|a_i|$ and applying induction.

For example, in the case of multiplication, the vanishing polynomial is

$$a(x) \cdot b(x) - res(x) - carry(x) \cdot q(x)$$

and the maximum absolute value of the coefficient we can attain is (for `l = 16`)

$$M = (2^{16} - 1) + 16 \cdot (2^{16} - 1)^2$$

In this case, we can set `WITNESS_OFFSET = 2^20` , and this can be shown by

$$M/2^{16} \le 2^{20}, \quad (M + 2^{20}) / 2^{16} \le 2^{20}$$

Soundness

The soundness can be shown by proving that the polynomial identity holds over not only $\mathbb{F}_p[x]$ but also $\mathbb{Z}[x]$. In that case, one can plug in $x = 2^{16}$ and read the result over $\mathbb{Z}$.

This can be also done by bounding the coefficients of the left/right hand side. For example, if

$$a(x) \cdot b(x) - res(x) - carry(x) \cdot q(x) = (x - 2^{16}) (w_{low}(x) + 2^{16} \cdot w_{high}(x)- \text{WITNESS\_OFFSET} \cdot (1 + x + \cdots + x^{2l-3}))$$

The left hand side has coefficients bounded with absolute value M and the right hand side has coefficients bounded with absolute value $(2^{16} + 1)2^{32}$. As we have $M + (2^{16} + 1)2^{32} < p$ for Goldilocks prime p, we see the two polynomials have to be equal in $\mathbb{Z}[x]$.

Uint

The `bit_operations` subfolder defines constraints for various bit operations over arrays of `BitRegister` s. There's also a byte decoding instruction that decomposes a byte as an array of 8 bits. However, most byte operations are intended to be handled with a pre-defined lookup table. The table iterates over all possible byte opcodes (`AND`, `XOR`, `SHR`, `ROT`, `NOT`, `RANGE`) and their inputs, compute their outputs, then digests the output into a single `u32` value, then inserts them into a lookup table. The digest is practically a byte-concatenation of the opcode, byte inputs and byte output - so for example for the `AND` opcode the digest for the operation `a AND b = c` is simply `OPCODE_AND + 256 * a + 256^2 * b + 256^3 * c` .

Later, when the byte operations are done via lookups, the digest is constrained to be computed correctly via `lookup_digest_constraint` and the digest is loaded into the vector of values to be looked up to. We will see that the current digest formula could lead to an underconstrained system, especially if all `ByteRegister` s aren't range checked immediately.

The `operations` subfolder defines bit operations done over arrays of `ByteRegister` s, and it also deals with addition over arrays of `ByteRegister` s as well.

Memory & Lookups

The memory model uses the "memory in the head" technique from Spartan (https://eprint.iacr.org/2019/550.pdf) and the lookup argument uses the technique derived from logarithmic derivatives (https://eprint.iacr.org/2022/1530.pdf).

In the memory, we have pointers that help us access the memory based on index shifts. The shifts can be in the form of an `ElementRegister` (so it's a variable) or a `i32` (so it's a constant). The `RawPointer` is the most fundamental pointer, consisting of a challenge `CubicRegister` as well as two shift variables, where one is `Option<ElementRegister>` and the other is `Option<i32>` . Its `eval` function simply returns `challenge + shift` where shift is the sum of two optional shift variables. A `RawSlice` is a slice of `RawPointer` s, which share the same `challenge` . A `Pointer` is a `RawPointer` with an `ArrayRegister` of challenges, and `Slice` is a slice of `Pointer` s, which share the same array of challenges. The additional challenges are used for the compression of the value that the pointer holds.

The `MemoryValue` trait for a `Register` comes with two functions - a `num_challenges()` function which denote the number of challenges it needs to compress the structure a `Register` is representing, and a `compress()` function, which, given the time of the write and the array of challenges, returns the compressed value inside a `CubicRegister` . With this in mind, the `PointerAccumulator` structure contains everything, the `RawPointer` , the `CompressedValue<F>` (which could either be a cubic element or just an element) and the final digest value as a `CubicRegister` . The constraints simply state that `digest = (challenge + shift) * compressed_value` computed in the extension field.

However, this formula leads to an issue - `digest = 0` whenever `compressed_value = 0` . This issue and its side effects are explored in vulnerability #10 in the findings section.

There are `Get` and `Set` instructions that deal with either fetching a value at a `RawPointer` and writing it in a `MemorySlice` or fetching a value at a `MemorySlice` and writing it at the `RawPointer` . These two instructions do not come with specific constraints, but rather are used with the trace writer which keeps track of the actual memory state.

The final memory checking argument is also done with a logarithmic derivative technique. The `LogEntry<T>` enum deals with various input/output cases based on whether there is a additional multiplicity represented with a `ElementRegister` . With a challenge β for the final check, the constraint system in `table/log_derivative/constraints.rs` deals with summing up all the $mult_i/(\beta - val_i)$ and accumulating it over all the rows.

The memory itself is handled with a `bus` structure. A `BusChannel` structure contains an `out_channel` which is a `CubicRegister` contains the final table accumulator value that accumulated all the entries over all table rows at the final row. A bus can have multiple

`BusChannel` s as well as global entries that may be input/output with multiplicity. The final constraint on the entire `Bus` constrains that the sum of the global accumulator and all the `out_channel` values from the `BusChannel` s sum to zero.

The very first `Bus` 's very first channel is used as the memory.

The lookup argument itself is very similar - first it accumulates all the `LogEntry` s from all the rows into a `digest` . It then accumulates the entries from the lookup table alongside the provided multiplicities using the logarithmic derivative technique and checks that the resulting table digest is equal to the sum of all value digests provided.

Machine

The audit for the machine folder deals with three major parts

- The correct application of Fiat-Shamir heuristic for proof generation and verification
- Correct handling of using multiple AIR instances at once
- The completeness and soundness of additional gadgets (elliptic curve, SHA hash function)

Fiat-Shamir and Challenge Generation

The Fiat-Shamir heuristic is done with a `Challenger` implementation. It keeps two buffers, one for inputs and one for outputs, and updates them based on the elements it observes. If a new element is observed, it clears the output buffer as it now contains results that did not take the new element into account. If sufficient number of elements are pushed into the input buffer, the sponge hash function is used to absorb the elements into the sponge state. If challenges need to be generated, the sponge state is squeezed. We note that the "overwrite mode" for sponge hash is used. There's also an in-circuit version of the `Challenger` .

The prover for a single AIR instance works as follows.

It first pushes the entire vector of public inputs to the challenger. Then, for each round, it

- generates the trace for that round
- computes the commitment for the traces
- pushes the global values generated from that round to the challenger
- pushes the merkle cap for the commitment to the challenger
- generates the challenges for the next round

then, to compute the quotient polynomials, it

- generates challenges to linearly combine all the constraints

Then, it pushes the quotient polynomial's commitment merkle cap to the challenger. Then, the challenge to select the coset for the FRI proof is generated. After that, the openings for the proof is pushed to the challenger, then the opening proof is generated with the challenger.

The internal logic and implementation within the FRI prover and verifier is out of scope.

Using multiple AIR instances: `BytesBuilder` and `EmulatedBuilder`

A `BytesBuilder` exists to have all the byte operations handled in a separate AIR. To do so, `ByteStark` struct has two STARKs - `stark` and `lookup_stark` .

The `lookup_stark` contains the byte lookup table as well as the table accumulation constraints, and the `stark` contains all the `ByteLookupOperations` and the values accumulation constraints. By sharing the memory state and using the same set of global values between the STARKs, one can delegate byte operations to the `lookup_stark` .

The prover works as follows

- push all public values to the challenger
- generate all execution traces as well as the lookup multiplicity data
- commit to all execution traces, for both `stark` and `lookup_stark`
- push both merkle tree caps for `stark` and `lookup_stark`
- generate AIR challenges and save them for both `stark` and `lookup_stark`
- generate extended traces while making sure that both are consistent
- commit all extended traces
- push the global values to the challenger
- push the extended trace merkle tree caps to the challenger
- prove each individual STARK with the challenger passed along

In the verification, the same `global_values` is used to verify the two STARKs.

The `EmulatedBuilder` is used to lookup all arithmetic columns and global arithmetic columns into a lookup table that contains `0, 1, ... 2^16 - 1` . Similar to `BytesBuilder` , two STARKs are used, with one STARK containing the lookup values and multiplicities.

Elliptic Curve Builder

The `scalar_mul_batch` function allows batch verification of `scalar * point = result` where each `point, scalar, result` comes from a public memory slice. We outline the constraints.

For easier explanation, assume that the scalar for the elliptic curve is 256 bits. Each instance of `scalar * point = result` takes 256 rows. A cycle of size 256 is generated so that the `BitRegister` `end_bit` can check whether the current instance ends at the current row. A `process_id` is a `ElementRegister` that denotes the index of the current row's instance.

Also, as the scalars are written as 8 limbs with each having 32 bits, a cycle of size 32 is generated so the `end_bit` can check whether the current limb ends at that row. Similarly, a `process_id_u32` is a `ElementRegister` that denotes the index of the current row's limb.

Also, a `result` register is allocated to keep track of intermediate results.

There are multiple pointer slices defined.

`temp_x_ptr`, `temp_y_ptr`

- `temp_{x,y}_ptr[i]` corresponds to the `i` th instance of `(point, scalar, result)`
- `temp_{x,y}_ptr[i]` gets written as `2^j * point[i]` at time `256 * i + j`
  - at the beginning `temp_{x,y}_ptr[i]` gets written as `point[i]` at time `256 * i`
  - at the `256 * i + j` th row
    - `temp_{x,y}_ptr[i]` gets loaded with time `256 * i + j`
    - the loaded point gets multiplied by 2, and the result gets written at `temp_{x,y}_ptr[i]` at time `256 * i + j + 1` if `end_bit` is false

`x_ptr`, `y_ptr`

- `{x,y}_ptr[i]` corresponds to the `i` th `result`
- at the beginning, `{x,y}_ptr[i]` gets freed at time `0` with value `result.{x,y}`
- `x_ptr[process_id]` is stored with multiplicity `end_bit` with value `result`
  - this guarantees that each `x_ptr` is stored exactly once, with the final `result` value

`limb_ptr`

- `limb_ptr[8 * i + j]` stores the `j` th limb of the `i` th `scalar` with multiplicity `32`
- `limb_ptr[process_id_u32]` is loaded at each row, leading to `32` loads
- the loaded limb is decomposed into bits, named `scalar_bit`

With these pointer slices, the main constraints implement the double-and-add algorithm.

A `is_res_unit` is used to check if the current intermediate result is a point at infinity.

- `is_res_unit = 1` at the first row
- `is_res_unit_next = (1 - sclalar_bit) * is_res_unit` if `end_bit` is false
- `is_res_unit_next = 1` if `end_bit` is true

The `result` is set to a point at infinity at the first row as well as when a new instance begins. The transition of `result` work as follows. The natural idea is to compute

- `result_next = result + temp` when `scalar_bit` is true
- `result_next = result` when `scalar_bit` is false

where the handling the case where `end_bit` is done separate.

However, since addition with point at infinity is an edge case, we avoid it as follows.

- `addend = is_res_unit ? 2 * temp : result`
- `sum = temp + addend`
- `res_plus_temp = is_res_unit ? temp : sum`
- `result_next = scalar_bit ? res_plus_temp : res`

We see that in the case where `is_res_unit` is true, `res_plus_temp` becomes `temp` without having any additions with point at infinity. In the case where `is_res_unit` is false, `res_plus_temp` becomes `res + temp` as we desired.

SHA Builder

A similar memory checking argument is used to implement the SHA2 hash functions.

While the precise preprocessing and processing steps are left to be implemented by the implementers of `SHAir` trait, the main logic that actually get/set the values to be processed from the pointer slices are written in `hash/sha/algorithm.rs` . In other words, while the exact formula for the message scheduling is implemented by files like `sha/sha256/air.rs` , the logic to load/store values from the message scheduling to the pointers is in `hash/sha/algorithm.rs` .

Each row deals with a single loop of the message scheduling and the main loop. We give a rough explanation based on SHA256, against the pseudocode at Wikipedia (https://en.wikipedia.org/wiki/SHA-2#Pseudocode).

The public inputs are

- `padded_chunks` : the data to be hashed
- `end_bits` : denotes whether to reset the hash state to initial state after the round
- `digest_bit` : denotes whether to declare the hash state as the result after the round
- `digest_indices` : the indices of the rounds where `digest_bit` is true

First, various slices are defined.

- `round_constants` simply puts the round constant values into a slice
- `shift_read_mult` puts the number of accesses for an index while loading `w[i-15], w[i-2], w[i-16], w[i-7]` in the preprocessing step
- `end_bit` and `digest_bit` are defined with the corresponding public inputs
- `is_dummy_slice` contains whether a round is a dummy round or not
- `w` is the slice that contains the `w` values - it is first stored with the padded chunks at their respective indices with multiplicity 1. A dummy index handles out of bound reads.

A `process_id` is allocated to denote the index of the current round. A `cycle_end_bit` is allocated to denote whether the current row is a final row of the current round. A `index` is allocated to denote the index of the current loop in the current round, so `index = row_index - 64 * process_id`. A `is_preprocessing` `BitRegister` is allocated to denote if the current loop is preprocessing - so it's `0` when `0 <= index < 16` and `1` when `16 <= index < 64`.

In the preprocessing, the first task is to fetch `w[i-15], w[i-2], w[i-16], w[i-7]`. To do so, the actual index to access the slice `w` is computed. If it's a dummy round, the dummy index should be accessed. If `is_preprocessing` is `0`, the preprocessing doesn't happen so the dummy index should be accessed. In the other cases, `i-15, i-2, i-16, i-7` is the correct actual index to access. With the loaded `w[i-15], w[i-2], w[i-16], w[i-7]`, the internal preprocessing step function appropriately computes the correct `w[i]` value.

Now we move on to storing this computed `w[i]` value. If it is a preprocessing loop or it's a dummy round, there's nothing at `w[i]` so a dummy index should be accessed. If not, then `w[i]` will be fetched. Therefore, the correct `w[i]` will be

- if `is_preprocessing` is true, computed from the preprocessing step function
- if `is_preprocessing` is false, read from `w` directly

If it's a dummy round, there is no read to write this `w` value into the `w` slice. If it's not a dummy round, this `w[i]` value is written to the slice with multiplicity `shift_read_mul[index]`. The final value of `w[i]` is kept as a register and used later. This way, in the processing step `w[i]` is directly used as a register.

In the processing step, a `state_ptr` slice is defined. This will store & free hash results when `digest_bit = 1`. The `i` th hash instance ends with the `digest_indices[i]` th cycle. Denote the result of this hash as `hash[i]`, which will be a public register array. Then,

- `state_ptr[j]` is freed with value `hash[i]`'s `j` th chunk with time `digest_indices[i]`

The round constant for the current index can be fetched from the `round_constant` slice, and `w[i]` can be directly used as a register. The values for `a, b, c, d, e, f, g, h` (denoted `vars`) and `h0, h1, h2, h3, h4, h5, h6, h7` (denoted `state`) are initialized at the first row. The computation of `vars_next` (i.e. the compression function main loop) is done in a `processing_step` function, and the addition of the compressed chunk to the current hash value to compute `state_next` is done in a `absorb` function.

The storing of the `state_next` onto the `state_ptr` should be done when the cycle ends, it's not a dummy round, and the `digest_bit` corresponding to the current `process_id` is `1`. In that case, the `state_next` is stored to the `state_ptr` with time `process_id`.

As for the state transitions with `vars_next` and `state_next`,

- if `cycle_end_bit` is false
  - corresponds to the case where it's still within the main loop
  - the next `var` should be `vars_next`

    - the next `state` should still be `state`
- if `cycle_end_bit` is true but `end_bit` is false
    - corresponds to the case where the main loop is over, but it's still hashing
    - the next `var` should be `state_next`
    - the next `state` should be `state_next`
- if `cycle_end_bit` is true and `end_bit` is true
    - corresponds to the case where the main loop and the hash is over
    - the next `var` should be `initial_hash`
    - the next `state` should be `initial_hash`

Codebase Overview - Plonky2x

uint

The uint folder works with the big integers. They are represented using 32-bit limbs, and overall computation on the integers are done with gadgets that work on those 32-bit limbs.

We highlight some of the more important constraints here. All of the range-checks for a variable to be within u32 range is done by decomposing the variable to 16 limbs of 2 bits each. Each limbs are checked to be within 2 bits via `x * (x - 1) * (x - 2) * (x - 3) == 0`.

add_many_u32

- adds at most 16 u32 values with a input carry
- constrains `output_carry * 2^32 + output_result = sum(input) + input_carry`
- constrains `output_carry` is at most 4 bits using the limbs
- constrains `output_result` is at most 32 bits using the limbs

subtract_u32

- computes `x - y - borrow` and outputs `result` and `borrow`
- constrains `x - y - borrow + 2^32 * output_borrow = output_result`
    - if `x, y` are u32 and `borrow` is a bit, then `x - y - borrow` is within `[-2^32, 2^32)` range, so valid `output_borrow` and `output_result` is guaranteed to exist
- constrains that `output_result` is at most 32 bits
- constrains that `output_borrow` is a bit

arithmetic_u32

- aims to compute `a * b + c` where `a, b, c` are u32
    - note `(2^32 - 1)^2 + (2^32 - 1) = 2^64 - 2^32 < 2^64 - 2^32 + 1 = p`
- outputs `output_low` and `output_high` and has an intermediate `inverse`

- constrains that `a * b + c = output_low + 2^32 * output_high`
- constrains that `output_low` , `output_high` are at most 32 bits
- constrains that `output_low * ((2^32 - 1 - output_high) * inverse - 1) == 0`
    - implies that either `output_low = 0` or `output_high != 2^32 - 1`
    - therefore, maximum value of `output_low + 2^32 * output_high` is `2^64 - 2^32 < 2^64 - 2^32 + 1 = p` , which shows that this doesn't overflow modulo Goldilocks

comparison

- has two parameters `num_bits` and `num_chunks` which lead to parameter `chunk_bits`
    - `num_bits` are the bit length of the values to be compared
    - `num_chunks` is the number of chunks the values are decomposed to
    - `chunk_bits` is the number of bits for each chunk
    - `chunk_bits = ceil_div(num_bits, num_chunks)`
- aims to compare `a` and `b` and return if `a <= b` is true
- constrains that `a` and `b` are decomposed to chunks correctly
    - constrains that the sum of the chunks is equal to `a` and `b` respectively
    - constrains that each chunk is within `[0, 2^chunk_bits)`
- iterating from the least significant chunk to the most, run
    - `(b_i - a_i) * inverse == 1 - equal_i`
    - `(b_i - a_i) * equal_i == 0`
    - `diff_so_far = equal_i * diff_so_far + (1 - equal_i) * (b_i - a_i)`
    - Here, `a_i, b_i` are two chunks of `a, b` respectively
    - The first two constraints show that
        - `equal_i = 1 <==> a_i = b_i`
        - `equal_i = 0 <==> a_i != b_i`
    - The final constraint show that
        - `diff_so_far` is the "most significant" difference so far
- decompose `2^chunk_bits + diff_so_far` into `chunk_bits + 1` bits
    - constrain that each bit is a bit via `x * (x - 1) == 0`
    - constrain that the binary sum is equal to `2^chunk_bits + diff_so_far`
- returns the most significant bit of `2^chunk_bits + diff_so_far`

merkle, mapreduce

There two merkle tree implementations in the merkle folder.

The `SimpleMerkleTree` is a merkle tree with

- leaf hash formula `sha256(leaf_bytes)`
- inner hash formula `sha256(left || right)`

- empty nodes filled with zeroes and hashed accordingly
  - this implies that the number of empty nodes affect the tree root

The `TendermintMerkleTree` is a merkle tree with RFC6962 specs, i.e.

- leaf hash formula `sha256(0x00 || leaf_bytes)`
- inner hash formula `sha256(0x01 || left || right)`
- empty nodes are filled with zeroes, but not mixed in for the hash
  - this implies that the number of emmpty nodes do not affect the tree root

The mapreduce functionality works with

- a `ctx` variable, which is common context for all computation
- a `MapFn` function, which takes `B` inputs and outputs a value and a Poseidon digest
- a `ReduceFn` function, which takes two outputs and reduces it to a single output

Given `B * 2^n` constant inputs, the mapreduce function aims to

- break the inputs into `2^n` chunks
- run `MapFn` on each chunks
- use a binary-tree like approach with parallelization to reduce all outputs to one

`MapFn` circuit

- reads `B` inputs and applies the map function
- takes all the variables from the inputs and Poseidon hashes them
- outputs the `ctx` variable and the output as well as the hash

`ReduceFn` circuit

- reads the two "child circuit" proofs
- verifies the two proofs in-circuit
- asserts that the two `ctx` values match
- applies the `ReduceFn` on the outputs and the common `ctx`
- Poseidon hashes the pair of the two hash from the two child instances
- returns the `ctx`, the reduced output, as well as the hash

The final circuit

- has a computed Poseidon result as a constant
- verifies the final circuit's proof in-circuit
- constrains that the `ctx` is the required one
- constrains that the hash digest is the expected one
- returns the final reduced output

ecc, hash

The two folders deal with delegating proofs to the curta prover.

In the case of ecc, by calling functions such as `curta_25519_scalar_mul` in `ecc/curve25519/ec_ops.rs`, a request is pushed onto the `EcOpAccelerator`, initializing a `AffinePointVariable<Ed25519>` in the process if the operation returns a point.

Upon building the entire circuit, the `curta_constrain_ec_op` function in `ecc/curve25519/curta/builder.rs` is called. Here, the witnesses for the requests in `EcOpAccelerator` are generated and the results are constrained to be equal to the witnesses. Note that this only constrains the results to be equal to the "raw witnesses", so the correctness of the witness itself is not checked yet - this is delegated to the curta prover.

Then, the requests and the responses are put together as a `Ed25519OpVariable` - then their values are read and sent to the `Ed25519Stark` in `ecc/curve25519/curta/proof_hint.rs`. The stark proof and the public inputs are written on the output stream, which is basically what's read on the `ecc/curve25519/curta/builder.rs` side then verified as well.

The curta STARK is built as follows - it only takes the request type data. In the case of add/decompress/valid checks, it directly uses the functionalities built in curta. In the case of scalar multiplication, the `scalar_mul_batch` function is used instead.

The verifier has to do two things -

- verify the curta STARK in-circuit
- check the consistency between the request input and the curta STARK public data
  - the request input is stored as variables already in `ec_ops`
  - the STARK public data can be read as variables via `read_proof_with_public_input`
    - the exact registers in the curta STARK can be read from `operations`

Regarding the hash, two hash functions BLAKE2 and SHA2 are the most complex.

The main logic is similar to the one of ecc - delegate the proof to curta, and compare the public inputs to the curta STARK with the requests within the plonky2x builder. One additional element we audited is on the padding - especially on the SHA2 side, where it's done in-circuit.

It is assumed that the requests itself are correctly constrained externally.

Findings

1. [High] Incorrect `WITNESS_OFFSET` leads to a completeness issue on `FpInnerProductInstruction`

The inner product instruction gets two vectors of `FieldRegister` s `a` and `b`, then aims to compute the inner product of the two vectors. This instruction is used in the ed25519 related computations, to take inner product of two vectors of length 2.

```
pub fn ed_add<E: EdwardsParameters>(
        &mut self,
        p: &AffinePointRegister<EdwardsCurve<E>>,
        q: &AffinePointRegister<EdwardsCurve<E>>,
    ) -> AffinePointRegister<EdwardsCurve<E>>
    where
        L::Instruction: FromFieldInstruction<E::BaseField>,
    {
        let x1 = p.x;
        let x2 = q.x;
        let y1 = p.y;
        let y2 = q.y;

        // x3_numerator = x1 * y2 + x2 * y1.
        let x3_numerator = self.fp_inner_product(&vec![x1, x2], &vec![y2, y1]);

        // y3_numerator = y1 * y2 + x1 * x2.
        let y3_numerator = self.fp_inner_product(&vec![y1, x1], &vec![y2, x2]);

        // f = x1 * x2 * y1 * y2.
        let x1_mul_y1 = self.fp_mul(&x1, &y1);
        let x2_mul_y2 = self.fp_mul(&x2, &y2);
        let f = self.fp_mul(&x1_mul_y1, &x2_mul_y2);

        // d * f.
        let d_mul_f = self.fp_mul_const(&f, E::D);

        // x3 = x3_numerator / (1 + d * f).
        let x3_ins = self.fp_den(&x3_numerator, &d_mul_f, true);

        // y3 = y3_numerator / (1 - d * f).
        let y3_ins = self.fp_den(&y3_numerator, &d_mul_f, false);

        // R = (x3, y3).
        AffinePointRegister::new(x3_ins.result, y3_ins.result)
    }
```

As other field instructions, the constraint system is built upon the formula

$$\sum_{i=1}^{n} a_i(x)b_i(x) - res(x) - carry(x)q(x) = (x - 2^{16})w(x)$$

However, in the case where $n = 2$, the maximum absolute value for the coefficients of the left hand side goes up to $(2^{16} - 1)^2 \cdot 16 \cdot 2$. Based on the completeness analysis in the code overview section, it's clear that the provably correct `WITNESS_OFFSET` value is `2^21` in this case. However, the `WITNESS_OFFSET` is set as `2^20`, as seen in `edwards/ed25519/params.rs`.

```
impl FieldParameters for Ed25519BaseField {
    const NB_BITS_PER_LIMB: usize = 16;
    const NB_LIMBS: usize = 16;
    const NB_WITNESS_LIMBS: usize = 2 * Self::NB_LIMBS - 2;
    const MODULUS: [u16; MAX_NB_LIMBS] = [
        65517, 65535, 65535, 65535, 65535, 65535, 65535, 65535, 65535, 65535,
        65535, 65535, 32767, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    ];
    const WITNESS_OFFSET: usize = 1usize << 20;

    fn modulus() -> BigUint {
        (BigUint::one() << 255) - BigUint::from(19u32)
    }
}
```

This implies that the ed25519 addition formula loses completeness.

```rust
    fn test_ed25519_add_fail() {
        type L = Ed25519AddTest;
        type SC = PoseidonGoldilocksStarkConfig;
        type E = Ed25519;

        let mut builder = AirBuilder::<L>::new();

        let p_pub = builder.alloc_public_ec_point();
        let q_pub = builder.alloc_public_ec_point();
        let _gadget_pub = builder.ec_add::<E>(&p_pub, &q_pub);

        let p = builder.alloc_ec_point();
        let q = builder.alloc_ec_point();
        let _gadget = builder.ec_add::<E>(&p, &q);

        let num_rows = 1 << 16;
        let (air, trace_data) = builder.build();
        let generator = ArithmeticGenerator::<L>::new(trace_data, num_rows);

        let x = BigUint::parse_bytes(
            b"5788367523335847815533809665734407736289112118965508746301431575456
            10,
        )
        .unwrap();
        let y = BigUint::parse_bytes(
            b"1640881932870819773037589667850624929038007064061241449739863549601
            10,
        )
        .unwrap();

        let p_int = AffinePoint::new(x, y);
        let q_int = p_int.clone();
        let writer = generator.new_writer();
        (0..num_rows).into_par_iter().for_each(|i| {
            writer.write_ec_point(&p, &p_int, i);
            writer.write_ec_point(&q, &q_int, i);
            writer.write_ec_point(&p_pub, &p_int, i);
            writer.write_ec_point(&q_pub, &q_int, i);
            writer.write_row_instructions(&generator.air_data, i);
        });

        writer.write_global_instructions(&generator.air_data);

        let stark = Starky::new(air);
        let config = SC::standard_fast_config(num_rows);
        let public = writer.public().unwrap().clone();

        // Generate proof and verify as a stark
        test_starky(&stark, &config, &generator, &public);

        // Test the recursive proof.
```

```
        test_recursive_starky(stark, config, generator, &public);
    }
```

To patch this issue, we recommend setting the `WITNESS_OFFSET` to `2^21` .

Fix Notes

Patched in this commit (https://github.com/succinctlabs/curta/pull/126/commits/e013d59c78d5323d68c677bd60c4accddd1bc392) by setting the `WITNESS_OFFSET` to `2^21` as recommended.

2. [Critical] `ShrCarry` operation is underconstrained

The `ShrCarry` operation gets `a, shift, result, carry` and aims to constrain that `result = a >> shift` and `carry = a % 2^shift` . To do so, it constrains that `digest = OPCODE_ROT + 256 * a + 256^2 * shift + 256^3 * (result + carry * 2^(8 - shift))` is within the lookup table. The concept behind this logic is that this basically forces `result + carry * 2^(8 - shift)` to be equal to the rotated value of `a` . However, constraining that `result + carry * 2^(8 - shift) = rot(a, shift)` is not sufficient to constrain `result = a >> shift` and `carry = a % 2^shift` . For example, values like `a = 255` , `shift = 2` , `result = 191` , `carry = 1` also pass all the constraints. As the correctness of `ShrCarry` operation is critical for logic in `uint/operations/shr.rs` and `uint/operations/rotate.rs` dealing with byte arrays, the logic regarding shifts & rotations of bytearrays are also underconstrained as well.

```
// uint/bytes/operations/value.rs
ByteOperation::ShrCarry(a, shift, result, carry) => {
    let a = a.eval(parser);
    let shift_val = parser.constant(AP::Field::from_canonical_u8(*shift));
    let carry = carry.eval(parser);
    let result = result.eval(parser);

    let mut c =
        parser.mul_const(carry, AP::Field::from_canonical_u16(1u16 << (8 - sh
    c = parser.add(result, c);

    let constraint = byte_decomposition(element, &[opcode, a, shift_val, c],
    parser.constraint(constraint);
}
```

```
// uint/operations/rotate.rs
pub fn set_bit_rotate_right<const N: usize>(
        &mut self,
        a: &ByteArrayRegister<N>,
        rotation: usize,
        result: &ByteArrayRegister<N>,
        operations: &mut ByteLookupOperations,
    ) where
        L::Instruction: From<ByteOperationInstruction>,
    {
        // ....
        let (last_rot, last_carry) = (self.alloc::<ByteRegister>(), self.allc
        let shr_carry = ByteOperation::ShrCarry(
            a_bytes_rotated[N - 1],
            bit_rotation as u8,
            last_rot,
            last_carry,
        );
        self.set_byte_operation(&shr_carry, operations);

        let mut carry = last_carry.expr();
        for i in (0..N - 1).rev() {
            let (shift_res, next_carry) =
                (self.alloc::<ByteRegister>(), self.alloc::<ByteRegister>());
            let shr_carry = ByteOperation::ShrCarry(
                a_bytes_rotated[i],
                bit_rotation as u8,
                shift_res,
                next_carry,
            );
            self.set_byte_operation(&shr_carry, operations);
            let expected_res = shift_res.expr() + carry.clone() * mult;
            self.set_to_expression(&result_bytes.get(i), expected_res);
            carry = next_carry.expr();
        }

        // Constraint the last byte with the carry from the first
        let expected_res = last_rot.expr() + carry.clone() * mult;
        self.set_to_expression(&result_bytes.get(N - 1), expected_res);
    }
```

We recommend to add more range checking constraints onto `result` and `carry`.

Fix Notes

This was fixed by adding both `result` and `carry` to the lookup table, in [this commit (https://github.com/succinctlabs/curta/pull/126/commits/a95b888bb69b98c7dc78b4db6fd4c0a29349a52c#diff-247b9bd0f8fbe5341157a7194b24752c883b88c92c57a38b9bbb966ca4dd105c)](https://github.com/succinctlabs/curta/pull/126/commits/a95b888bb69b98c7dc78b4db6fd4c0a29349a52c#diff-247b9bd0f8fbe5341157a7194b24752c883b88c92c57a38b9bbb966ca4dd105c).

The table itself is now precomputed, and the table's merkle caps are checked to be equal to this precomputed merkle cap. This verifies that the table itself was generated correctly.

3. [High] Byte Lookup's `lookup_digest` uses a fixed constant to combine values

In a way, the byte lookup digest formula `opcode + 256 * a + 256^2 * b + 256^3 * c` is an attempt to commit to the vector `(opcode, a, b, c)`. This method works in the case where we can guarantee that `a, b, c` are within byte range - in this case a digest corresponds to a unique `(opcode, a, b, c)`. However, in many cases this is not the case.

First, even when `ByteRegister` s are allocated, they are not directly range checked via decomposition into 8 bits. Therefore, one can prove that given `a, b`, one can prove that `a AND b = c` where `OPCODE::AND + 256 * a + 256^2 * b + 256^3 * c = OPCODE::XOR + 256 * 1 + 256^2 * 1 + 256^3 * 0`, **even in the case where `c` is not within byte range**.

In a way, this issue comes from the fact that the coefficients for the linear combination of `(opcode, a, b, c)` used to compute the digest is a fixed constant. In a standard vector lookup, a challenge γ is derived via Fiat-Shamir (after commiting to all relevant lookup instances) then the linear combination is done with consecutive powers of γ. Implementing the byte operation lookups in this fashion would resolve the issue without additional range checks.

The other method to resolve this issue is to strictly enforce that all `ByteRegister` go through a range check. This range check should not be based on the lookup table (as the lookup itself assumes values to be within byte range), but should be done via bitwise decomposition.

Fix Notes

This was fixed by using the Reed-Solomon footprint instead - see this commit (https://github.com/succinctlabs/curta/pull/126/commits/a95b888bb69b98c7dc78b4db6fd4c0a29349a52c) as well as this commit (https://github.com/succinctlabs/curta/pull/126/commits/eb7c2610705113fd83efbee8ad288eb73c8e3dcb). Combined with the fix from vulnerability #2, a challenge γ and their powers $1, \gamma, \cdots, \gamma^4$ are used to combine the vector `(OPCODE, a, b, c, d)`.

4. [Informational] `FpDenInstruction` may output arbitrary `result` in the case of `0/0` division

In the case of a standard division, to compute $a/b \pmod{q}$ the instruction checks two things

- $b \cdot inv \equiv 1 \pmod{q}$
- $a \cdot inv \equiv res \pmod{q}$

which is sufficient to enforce that b is invertible.

However, in the `FpDenInstruction`, there's no additional check for zero division. For example, if `sign = 1`, then to compute $a / (b + 1) \pmod{q}$, one simply checks that

$$(b + 1) \cdot res - a \equiv 0 \pmod{q}$$

which allows arbitrary res value when $b \equiv -1 \pmod{q}$.

In the codebase, `FpDenInstruction` is used to compute the addition formula for twisted edward curve addition. Thankfully, in the case for ed25519, as the addition formula is complete. Therefore, if the two points that's being added is guaranteed to be valid ed25519 points then there's no need for extra verification that a zero division is taking place.

However, in the case where `FpDenInstruction` is used for some other needs, then one needs to take into consideration that the instruction allows arbitrary outputs for `0/0` division.

We recommend writing additional documentation to notify this to the users of the library.

Fix Notes

Added documentation in this commit (https://github.com/succinctlabs/curta/pull/126/commits/288b298bbd933769f3f244b09698d3fc02d6104a), this commit (https://github.com/succinctlabs/curta/pull/126/commits/f9e08fa5db35f38a6308000cc4ed8ac3abcd6d10), and this commit (https://github.com/succinctlabs/curta/pull/126/commits/b6cb3c4698aca56cdf6c87c930e5f1dc4bc35236) as recommended.

5. [Critical] `LogLookupValues`'s `digest` is underconstrained, leading to arbitrary lookups

To register lookup values for a table, it creates three digests - a `local_digest` for the trace values being looked up, a `global_digest` for global values being looked up, and a `digest`, a sum of `local_digest` and `global_digest`. This `digest` value is pushed to the `values_digest` vector of the table.

```
pub(crate) fn new_lookup_values<L: AirParameters<Field = F, CubicParams = E>>
        &mut self,
        builder: &mut AirBuilder<L>,
        values: &[T],
    ) -> LogLookupValues<T, F, E> {
        let mut trace_values = Vec::new();
        let mut public_values = Vec::new();

        for value in values.iter() {
            match value.register() {
                MemorySlice::Public(..) => public_values.push(LogEntry::input
                MemorySlice::Local(..) => trace_values.push(LogEntry::input(*
                MemorySlice::Next(..) => unreachable!("Next register not supp
                MemorySlice::Global(..) => public_values.push(LogEntry::input
                MemorySlice::Challenge(..) => unreachable!("Cannot lookup cha
            }
        }

        let row_accumulators =
            builder.alloc_array_extended::<CubicRegister>(trace_values.len()
        let global_accumulators =
            builder.alloc_array_global::<CubicRegister>(public_values.len() /
        let log_lookup_accumulator = builder.alloc_extended::<CubicRegister>(

        let digest = builder.alloc_global::<CubicRegister>();
        let global_digest = Some(builder.alloc_global::<CubicRegister>());

        self.values_digests.push(digest);

        LogLookupValues {
            challenge: self.challenge,
            trace_values,
            public_values,
            row_accumulators,
            global_accumulators,
            // log_lookup_accumulator,
            local_digest: log_lookup_accumulator,
            global_digest,
            digest,
            _marker: PhantomData,
        }
    }
```

The lookup value constraints are the `ValuesLocal` and `ValuesGlobal` constraints, which proves the `local_digest` and `global_digest` are correctly computed.

```
impl<F: Field, E: CubicParameters<F>> LogLookupValues<ElementRegister, F, E>
    pub(crate) fn register_constraints<L: AirParameters<Field = F, CubicParam
        &self,
        builder: &mut AirBuilder<L>,
    ) {
        builder.constraints.push(Constraint::lookup(
            LookupConstraint::<ElementRegister, _, _>::ValuesLocal(self.clone
        ));
        if self.global_digest.is_some() {
            builder.global_constraints.push(Constraint::lookup(
                LookupConstraint::<ElementRegister, _, _>::ValuesGlobal(self.
            ));
        }
    }
}
```

On the other side, the table constraints are that the table logarithmic derivative is correctly computed, and that the sum of values inside `values_digest` is equal to the table digest.

```
pub fn constrain_element_lookup_table(
        &mut self,
        table: LogLookupTable<ElementRegister, L::Field, L::CubicParams>,
    ) {
        // insert the table to the builder
        self.lookup_tables.push(LookupTable::Element(table.clone()));

        // Register digest constraints between the table and the lookup value
        self.global_constraints.push(Constraint::lookup(
            LookupConstraint::<ElementRegister, _, _>::Digest(
                table.digest,
                table.values_digests.clone(),
            )
            .into(),
        ));

        // Register the table constraints.
        self.constraints
            .push(Constraint::lookup(LookupConstraint::Table(table).into()));
    }
```

```
LookupConstraint::Digest(table_digest, element_digests) => {
    let table = table_digest.eval_cubic(parser);
    let elements = element_digests
        .iter()
        .map(|b| b.eval_cubic(parser))
        .collect::<Vec<_>>();
    let mut elem_sum = parser.zero_extension();
    for e in elements {
        elem_sum = parser.add_extension(elem_sum, e);
    }
    let difference = parser.sub_extension(table, elem_sum);
    parser.constraint_extension_last_row(difference);
}
```

However, there is no constraint that `digest = local_digest + global_digest` . This is implemented in `LookupConstraint::ValuesDigest` , but this is never actually registered to the list of constraints. This enables the attacker to put any value at the `digest` , so arbitrary values can be looked up. This breaks the entire lookup argument.

We recommend to register the `LookupConstraint::ValuesDigest` constraint.

Fix Notes

The constraint registration is added in this commit (https://github.com/succinctlabs/curta/pull/126/commits/42416c59145c44914a65704b9e96180c61255d1b) and this commit (https://github.com/succinctlabs/curta/pull/126/commits/f1408b6d0047187e070bc2c505aa7411796631ae) as recommended.

6. [Low] Underflow in `rotate_left` leads to incorrect constraints

The `rotate_left` function is given the bit decomposition of the shift, and then uses it to compute the result iteratively. If the `k` th bit of the shift is `1` , then the result should be shifted by `2^k` once again. If the `k` th bit of the shift is `0` , then the result should be as it is.

This is done with a select operation as follows

```rust
for (k, bit) in b.into_iter().enumerate() {
    // Calculate temp.right_rotate(2^k) and set it to result if bit = 1
    let num_shift_bits = 1 << k;

    let res = if k == m - 1 {
        *result
    } else {
        self.alloc_array::<BitRegister>(n)
    };

    for i in 0..n {
        self.set_select(
            &bit,
            &temp.get((i - num_shift_bits) % n),
            &temp.get(i),
            &res.get(i),
        );
    }
    temp = res;
}
```

Note the `(i - num_shift_bits) % n` - this may be underflow. In all usage inside the curta codebase, `n` was a power of 2, leading to no differences even the underflow happens.

However, if `n` is not a power of 2, the underflow leads to an incorrect constraint.

We recommend adding either

- writing additional documentation and input validation so `n` is a power of `2`
- or, modifying the code so to prevent any potential underflows and overflows

We also note that if `n` is not a power of 2 and the shift is very large, then that also leads to an incorrect constraint as `num_rotate_bits` overflows to `0` at large `k`. See code below.

```
for (k, bit) in b.into_iter().enumerate() {
    // Calculate temp.right_rotate(2^k) and set it to result if bit = 1
    let num_rotate_bits = 1 << k;

    let res = if k == m - 1 {
        *result
    } else {
        self.alloc_array::<BitRegister>(n)
    };

    for i in 0..n {
        self.set_select(
            &bit,
            &temp.get((i + num_rotate_bits) % n),
            &temp.get(i),
            &res.get(i),
        );
    }
    temp = res;
}
```

Fix Notes

The fix is done in [this commit](https://github.com/succinctlabs/curta/pull/126/commits/e5859eaa39be18f88351072544c7255a1893c8ce) (https://github.com/succinctlabs/curta/pull/126/commits/e5859eaa39be18f88351072544c7255a1893c8ce) and [this commit](https://github.com/succinctlabs/curta/pull/126/commits/f9e08fa5db35f38a6308000cc4ed8ac3abcd6d10) (https://github.com/succinctlabs/curta/pull/126/commits/f9e08fa5db35f38a6308000cc4ed8ac3abcd6d10). We provide some additional insight below.

- There remains the potential overflow from having large `k` in `rotate.rs`
- There is also a possible over/underflow in `shift.rs` - but it doesn't affect anything
    - in `set_shr` and `set_shl`, if `num_shift_bits > n`, then `temp.get()` will access an index larger than `n` - leading to a failure

These facts should considered by the users of the library. In general, the users of the library should check that parameters such as `n` and `m` are appropriate for usage.

7. [Low] `register_lookup_values` for `CubicRegister` doesn't push to `builder.lookup_values`

```
impl<F: Field, E: CubicParameters<F>> LogLookupTable<ElementRegister, F, E> {
    pub fn register_lookup_values<L: AirParameters<Field = F, CubicParams = E
        &mut self,
        builder: &mut AirBuilder<L>,
        values: &[ElementRegister],
    ) -> LogLookupValues<ElementRegister, F, E> {
        let lookup_values = self.new_lookup_values(builder, values);
        lookup_values.register_constraints(builder);
        builder
            .lookup_values
            .push(LookupValues::Element(lookup_values.clone()));
        lookup_values
    }
}

impl<F: Field, E: CubicParameters<F>> LogLookupTable<CubicRegister, F, E> {
    pub fn register_lookup_values<L: AirParameters<Field = F, CubicParams = E
        &mut self,
        builder: &mut AirBuilder<L>,
        values: &[CubicRegister],
    ) {
        let lookup_values = self.new_lookup_values(builder, values);
        lookup_values.register_constraints(builder);
    }
}
```

In `LogLookupTable<CubicRegister, F, E>`, the `lookup_values` is not pushed to the `builder.lookup_values`. This affects the trace generation, and would lead to the lookup values not being filled to the trace. However, this went unnoticed as it was never used.

Fix Notes

The missing line of code is added in [this commit](https://github.com/succinctlabs/curta/pull/126/commits/5c23768f47bbe230003868b833b3ff40caa21da0) (https://github.com/succinctlabs/curta/pull/126/commits/5c23768f47bbe230003868b833b3ff40caa21da0).

8. [Critical] bitwise decomposition in `ec/scalar.rs` underconstrained

An important part of the elliptic curve instruction is to decompose the scalar into bits which would later be used for the double-and-add algorithm. To do so, the scalar is first input as an array of limbs of 32 bits. The `start_bit` and `end_bit` is assumed to denote the starting/ending bit for a cycle of length 32. In that case, the constraints are as follows.

- `start_bit * (bit_accumulator - limb) = 0`
- `(1 - end_bit) * (2 * bit_accumulator_next - bit_accumulator + bit) = 0`

In other words, it asserts that on the starting bit the `bit_accumulator` is equal to the `limb`. Then, if it's not the final bit, it asserts that `bit_accumulator = 2 * bit_accumulator_next + bit`. However, there's one constraint missing - it must constrain that if `end_bit` is true, then `bit_accumulator = bit`. Without this constraint, the system is underconstrained.

```
// chip/ec/scalar.rs
impl<AP: AirParser> AirConstraint<AP> for LimbBitInstruction {
    fn eval(&self, parser: &mut AP) {
        // Assert the initial valuen of `bit_accumulator` at the begining of
        // are presented in little-endian order, the initial value of `bit_ac
        // of the limb register at the beginning of the cycle. This translate
        //    `start_bit * (bit_accumulator - limb) = 0`
        let bit_accumulator = self.bit_accumulator.eval(parser);
        let start_bit = self.start_bit.eval(parser);
        let limb_register = self.limb.eval(parser);
        let mut limb_constraint = parser.sub(bit_accumulator, limb_register);
        limb_constraint = parser.mul(start_bit, limb_constraint);
        parser.constraint(limb_constraint);

        // Assert that the bit accumulator is summed correctly. For that purp
        // every row other than the last one in the cycle (determined by end_
        //     `bit_accumulator_next = (bit_accumulator - bit) / 2.`
        //
        // This translates to the constraint:
        //     `end_bit.not() * (2 * bit_accumulator_next - bit_accumulator +
        let bit = self.bit.eval(parser);
        let end_bit = self.end_bit.eval(parser);
        let one = parser.one();
        let not_end_bit = parser.sub(one, end_bit);
        let mut constraint = self.bit_accumulator.next().eval(parser);
        constraint = parser.mul_const(constraint, AP::Field::from_canonical_u
        constraint = parser.sub(constraint, bit_accumulator);
        constraint = parser.add(constraint, bit);
        constraint = parser.mul(not_end_bit, constraint);
        parser.constraint_transition(constraint);
    }
}
```

Fix Notes

The missing constraint was added in this commit (https://github.com/succinctlabs/curta/pull/126/commits/0df2a6a33509ad8a46738fbc5104eedb02a69050) as recommended.

9. [High] Decompression of `ed25519` points allow sign change

The `ed25519_decompress` function allows the `AirBuilder` to turn a compressed point into a decompressed point. A compressed point register consists of a `BitRegister` called `sign` and a `FieldRegister<Ed25519BaseField>` called `y`. This `y` will be the `y`-coordinate of the final ed25519 point, and the `sign` would determine which `x` value the point would have. If `sign = 0`, then the `x` value that is even and within `[0, q)` would be selected. If `sign = 1`, then the `x` value that is odd and within `[0, q)` would be selected.

The core of the implementation is as follows - it first computes in-circuit the correct value of `x^2`, named `u_div_v` in the code. Then, it uses a `ed25519_sqrt` function to compute `x`, the square root of `u_div_v` that is even. Then, it selects between `x` and `fp_sub(0, x)` based on `sign` to compute the final `x`-coordinate value. Finally, it returns `(x, y)`.

```
let zero_limbs = vec![0; num_limbs];
let zero_p = Polynomial::<L::Field>::from_coefficients(
    zero_limbs
        .iter()
        .map(|x| L::Field::from_canonical_u16(*x))
        .collect_vec(),
);
let zero: FieldRegister<Ed25519BaseField> = self.constant(&zero_p);

let yy = self.fp_mul::<Ed25519BaseField>(&compressed_p.y, &compressed_p.y);
let u = self.fp_sub::<Ed25519BaseField>(&yy, &one);
let dyy = self.fp_mul::<Ed25519BaseField>(&d, &yy);
let v = self.fp_add::<Ed25519BaseField>(&one, &dyy);
let u_div_v = self.fp_div::<Ed25519BaseField>(&u, &v); // this is x^2

let mut x = self.ed25519_sqrt(&u_div_v);
let neg_x = self.fp_sub::<Ed25519BaseField>(&zero, &x);
x = self.select(&compressed_p.sign, &neg_x, &x);

AffinePointRegister::<EdwardsCurve<Ed25519Parameters>>::new(x, compressed_p.y
```

Denote `r` as the correct return value of `u_div_v` - the square root that is even.

The `ed25519_sqrt` function does two things. It first checks that `result * result == input (mod q)` using the `FpMulInstruction`. Then, it witnesses 15 bits `b_1 ~ b_15` to show that the first 16-bit limb of `result` is equal to `[b_1b_2....b_15]0` in binary.

This can be exploited as follows. Note that `r' = 2q - r` is also a valid square root of `u_div_v` and is also even. It is also within bounds, as `2q - r < 2^256` and any value within `2^256` can be encoded with a degree 15 polynomial. To check that `r' * r' == u_div_v (mod q)`, one needs to find `0 <= carry < 2^256` such that `r' * r' - carry * q = u_div_v`, but this is possible as long as `(2q - r)^2 <= 2^256 * q` or `r >= 2q - 2^128 * sqrt(q) ~ (2 - sqrt(2))q`. Therefore, for significant portion of `r`, using `r' = 2q - r` instead is possible.

Now to compute `neg_x = fp_sub(0, x)`, it's sufficient to show `x + neg_x - carry * q = 0`. Therefore, with `x = r'`, one can simply use `neg_x = r` with `carry = 2`.

In the case where `sign = 1`, the `neg_x` will be chosen, which would be `r`. The intended value here would be the odd square root, which is `p - r`. Since `r` is chosen, additional range checks (such as checking the value is less than `q`) would not work as well.

We recommend to check the range of the returned value of `ed25519_sqrt`.

Fix Notes

The idea now is to make the function *return* the value returned by `ed25519_sqrt` as well - and warning that all callers to the `ed25519_decompress` function should check that this returned value is between `0` and the `MODULUS`, to ensure soundness. The handling of this extra check is to be added in Plonky2x. The changes in Curta are done in [this commit](https://github.com/succinctlabs/curta/pull/126/commits/2040524f6f0df255d7a7881468befaaf07311866) (https://github.com/succinctlabs/curta/pull/126/commits/2040524f6f0df255d7a7881468befaaf07311866).

The changes in plonky2x is done in [this commit](https://github.com/succinctlabs/succinctx/pull/331/commits/b85a0c7368f9d814805a5d9b40af4c97565386c7) (https://github.com/succinctlabs/succinctx/pull/331/commits/b85a0c7368f9d814805a5d9b40af4c97565386c7) - now the connected variables in plonky2x is initialized with `init` instead of `init_unsafe`, so it is range checked. To be more precise, now the `root` variable as well as the `x, y` points from the decompressed result are all in range.

10. [Medium] The digest formula for memory leads to collisions between indices and different slices in edge cases

To recall, if there are multiple slices for a type `V`, the challenges are generated as follows.

- Each slice has a main `challenge`
- Each slice has compression challenges - the number is based on the type `V`
- For all the slices and the lookups and all, the challenge `beta` is generated

Here, the digest formula is `(challenge + shift) * compressed_value`, and the logarithmic derivative is used with the final challenge `beta`. Therefore, one needs to consider whether

- each digest corresponds to a unique slice (`challenge`), `shift` and `compressed_value`
- each `compressed_value` corresponds to a unique `value` and `time`

with high probability based on Fiat-Shamir.

First, the digest may not correspond to a unique `(challenge, shift, compressed_value)`. This is due to the case where `compressed_value = 0`. In all the other case, it's simple to prove that different set of `(challenge, shift, compressed_value)` would lead to a different `digest` with a high probability, as long as the `challenge` is generated via Fiat-Shamir.

However, in the case of same `compressed_value = 0`, any set of `challenge` or `shift` leads to the digest being zero. This allows the following memory access to work.

```
on the same slice

store value 1 at time 0 with multiplicity 1 at index 1
store value 0 at time 0 with multiplicity 1 at index 2 (compressed_value = 0)
free memory with value 1 at time 0 with multiplicity 1 at index 1
free memory with value 0 at time 0 with multiplcitiy 1 at index 1 (compressed_valu
```

```
on two different slices

store value 0 at time 0 with multiplicity 1 at index 1 at slice A (compressed_valu
free memory with value 0 with time 0 with multiplicity 1 at index 2 at slice B (co
```

This should be avoided. One method to do so is to use the digest formula `digest = compressed_value + time * challenge + index * challenge^2 + challenge^3`. This way, a `digest` corresponds to a single `(compressed_value, time, index, challenge)`.

Notice the separation of `compressed_value` and `time` in the above formula. In the current implementation, the value compression included both the value itself and the time.

```rust
impl MemoryValue for ElementRegister {
    fn num_challenges() -> usize {
        0
    }

    fn compress<L: crate::chip::AirParameters>(
        &self,
        builder: &mut AirBuilder<L>,
        ptr: RawPointer,
        time: &Time<L::Field>,
        _: &ArrayRegister<CubicRegister>,
    ) -> CubicRegister {
        let value = CubicElement([time.expr(), self.expr(), L::Field::ZERO.ir
        ptr.accumulate_cubic(builder, value)
    }
}
```

Notice that the value and time is simply combined as `CubicElement([time, value, 0])`.

This leads to other interesting observations - for example, the `MemoryValue` for `U32Register` is `CubicElement([value, time, 0])`, and for `U64Register` is `CubicElement([limb1, limb2, time])`. This implies that the `compressed_value` for `U32Register` with `value = 10` and `time = 10` is equal to the `compressed_value` for `U64Register` with `limb1 = 10`, `limb2 = 10` and `time = 0`. While this is not an immediate issue, it's an observation that is worth.

Fix Notes

The formula is now `compressed_value + shift * challenge + challenge^2`. This forces two equal digests to have the same `(compressed_value, shift, challenge)`. On the condition that a single `compressed_value` corresponds to a single `(value, time)`, this is fine.

This change is implemented in this commit (https://github.com/succinctlabs/curta/pull/126/commits/11f8c3e6bccd50f5c09c9c6de7dad3363600f2af).

11. [High] Uint allows remainder to be equal to the divisor

The standard constraints for division and remainder results when dividing `a` by `b` is

\[a = b \cdot q + r, \quad 0 \le r < b\]

where `q`, `r` are provided directly by witnesses rather than in-circuit computation.

```
fn _div_rem_biguint(
        &mut self,
        a: &BigUintTarget,
        b: &BigUintTarget,
        div_num_limbs: usize,
    ) -> (BigUintTarget, BigUintTarget) {
        let b_len = b.limbs.len();
        let div = self.add_virtual_biguint_target_unsafe(div_num_limbs);
        let rem = self.add_virtual_biguint_target_unsafe(b_len);

        self.add_simple_generator(BigUintDivRemGenerator::<F, D> {
            a: a.clone(),
            b: b.clone(),
            div: div.clone(),
            rem: rem.clone(),
            _phantom: PhantomData,
        });

        range_check_u32_circuit(self, div.limbs.clone());
        range_check_u32_circuit(self, rem.limbs.clone());

        let div_b = self.mul_biguint(&div, b);
        let div_b_plus_rem = self.add_biguint(&div_b, &rem);
        self.connect_biguint(a, &div_b_plus_rem);

        let cmp_rem_b = self.cmp_biguint(&rem, b);
        self.assert_one(cmp_rem_b.target);

        (div, rem)
    }
```

However, in the circuit in `uint/num/biguint/mod.rs`, the checks are

$$a = b \cdot q + r, \quad 0 \le r \le b$$

which allows $r = b$. This allows cases like `(a, b, q, r) = (6, 2, 2, 2)`.

```
fn test_biguint_div_rem_fail() {
        const D: usize = 2;
        type C = PoseidonGoldilocksConfig;
        type F = <C as GenericConfig<D>>::F;
        let mut rng = OsRng;

        let mut x_value = BigUint::from_u64(rng.gen()).unwrap();
        let mut y_value = BigUint::from_u64(rng.gen()).unwrap() * x_value.clo
        if y_value > x_value {
            (x_value, y_value) = (y_value, x_value);
        }
        let (expected_div_value, expected_rem_value) = x_value.div_rem(&y_val
        let expected_div_value = expected_div_value - BigUint::from_u64(1 as
        let expected_rem_value = expected_rem_value + y_value.clone();

        let config = CircuitConfig::standard_recursion_config();
        let pw = PartialWitness::new();
        let mut builder = CircuitBuilder::<F, D>::new(config);

        let x = builder.constant_biguint(&x_value);
        let y = builder.constant_biguint(&y_value);
        let (div, rem) = builder.div_rem_biguint(&x, &y);

        let expected_div = builder.constant_biguint(&expected_div_value);
        let expected_rem = builder.constant_biguint(&expected_rem_value);

        builder.connect_biguint(&div, &expected_div);
        builder.connect_biguint(&rem, &expected_rem);

        let data = builder.build::<C>();
        let proof = data.prove(pw).unwrap();
        data.verify(proof).unwrap()
    }
```

Fix Notes

The constraints now validate that `b <= rem` is false, so `rem < b`. Added in [this commit (https://github.com/succinctlabs/succinctx/pull/331/commits/1b7a2e3eafc39c0ac3c8fb091fcdf90fd3943ea7)](https://github.com/succinctlabs/succinctx/pull/331/commits/1b7a2e3eafc39c0ac3c8fb091fcdf90fd3943ea7).

12. [High] EC point validation skipped in plonky2x

```
impl Ed25519CurtaOp {
    pub fn new<L: AirParameters<Instruction = Ed25519FpInstruction>>(
        builder: &mut AirBuilder<L>,
        reuest_type: &EcOpRequestType,
    ) -> Self {
        match reuest_type {
            // ...
            EcOpRequestType::IsValid => {
                let point = builder.alloc_public_ec_point();
                Self::IsValid(point) // ##### <- no ed_assert_valid here? ###
            }
        }
    }
}
```

As seen by the code above, in the case where the request is to validate that a point is indeed on the ed25519 curve, the curta builder doesn't actually add the constraints to the curta STARK. Therefore, this validation is actually never done, allowing incorrect points to pass.

Fix Notes

This check is added in [this commit](https://github.com/succinctlabs/succinctx/pull/305/commits/c0a324582acdf7a5dcea4bdc0a7f2b126ff13e37) (https://github.com/succinctlabs/succinctx/pull/305/commits/c0a324582acdf7a5dcea4bdc0a7f2b126ff13e37) as recommended.

13. [Critical] `t_values` not connected in `blake2`

Recall that plonky2x's blake2 works by

- delegating the proof to curta
- verifying the curta proof in plonky2x in-circuit
- verifying that the public inputs in curta is equal to the instances in plonky2x

The issue here is that the `t_values`, one of the public inputs for `blake2`, is not connected between plonky2x and curta. This allows the curta prover to use a different `t_values`.

`t_values` correspond to the number of elements hashed so far in blake2. It is also the only public input for curta that contains the information regarding the exact length of the input bytearray. `t_values` is one of the inputs to the blake2 compression function - so even with the same padded chunks, a different `t_values` would lead to a different hash.

```
// hash/blake2/stark.rs
pub fn verify_proof(
    &self,
    builder: &mut CircuitBuilder<L, D>,
    proof: ByteStarkProofVariable<D>,
    public_inputs: &[Variable],
    blake2b_input: BLAKE2BInputData,
) {
    // Verify the stark proof.
    builder.verify_byte_stark_proof(&self.stark, proof, public_inputs);

    // Connect the public inputs of the stark proof to it's values.

    // Connect the padded chunks.
    // ...

    // Connect end_bits.
    // ...

    // Connect digest_bits.
    // ...

    // Connect digest_indices.
    // ...

    // Connect digests.
    // ...
}
```

Fix Notes

This is fixed in [this commit](https://github.com/succinctlabs/succinctx/pull/344/commits/9b1c0d26feace85010d8047c82b27a0b4e92dcbd) (https://github.com/succinctlabs/succinctx/pull/344/commits/9b1c0d26feace85010d8047c82b27a0b4e92dcbd) by connecting the `t_values` .

14. [High] `eddsa.rs` doesn't check `sig.s < scalar_order` leading to signature malleability

`eddsa.rs` deals with eddsa signature verification. The overall logic works as follows -

- computes `H(sig.r || pubkey || msg)` via SHA512
- computes `rem = H(sig.r || pubkey || msg) modulo l` where `l = scalar_order`
- verifies `sig.s * generator = sig.r + rem * pubkey`

Referring to papers such as [CGN20] (https://eprint.iacr.org/2020/1244.pdf) (Section 3) - we add some observations.

- There is no check that `0 <= sig.s < l` - which leads to signature malleability

- There is no check for non-canonical embeddings in `sig.r`

Both are important, but the first one is especially important to prevent signature malleability.

Fix Notes

Both are fixed in [this commit](https://github.com/succinctlabs/succinctx/pull/331/commits/b85a0c7368f9d814805a5d9b40af4c97565386c7) (https://github.com/succinctlabs/succinctx/pull/331/commits/b85a0c7368f9d814805a5d9b40af4c97565386c7). The `0 <= sig.s < l` check is directly added, and now from the additional range check from the decompression algorithm, it is verified that `sig.r` is indeed canonically embedded. For `sig.r` to be non-canonically embedded, the `y` value inside `sig.r` must be no less than `p`. However, in the decompression logic in curta, the returned `y` value of the resulting ed25519 point is just the input `y` value, with no operations done over it. Therefore, the range check added in plonky2x for the fix of vulnerability #9 forces that the `y` value in the compressed point `sig.r` must be within `[0, p)` range, as we desired.

Note that, in general, in plonky2x and in curta, it is the caller's responsibility to constrain that the scalar for the scalar multiplication is less than the group order.

15. [Medium] `list_lte_circuit` 's range check is not sound

The specification for `list_lte_circuit` claims that it performs a range check that each limb is at most `num_bits` bits. To do so, it uses a `ComparisonGate`.

```
/// Computes the less than or equal to operation between a and b encoded as b
///
/// Note that this function performs a range check on its inputs.
pub fn list_lte_circuit<F: RichField + Extendable<D>, const D: usize>(
    builder: &mut CircuitBuilder<F, D>,
    a: Vec<Target>,
    b: Vec<Target>,
    num_bits: usize,
) -> BoolTarget {
```

As mentioned at the code overview, `ComparisonGate` also has a parameter `chunk_bits` and `num_chunks` - these are used to constrain that each chunk is at most `chunk_bits` bits, and the chunks accumulate to be equal to the compared values.

However, the formula between the two is `chunk_bits = ceil_div(num_bits, num_chunks)`. This is a problem - the range check actually checks that the values are at most `chunk_bits * num_chunks` bits, and this value may be larger than `num_bits`.

For example, if `num_bits = 33, chunk_bits = 2, num_chunks = 17`, the values will be range checked to `34` bits, while the specifications say that they will be range checked to `33` bits.

A good enforcement to be added is `num_bits % chunk_bits == 0`.

Fix Notes

The `num_bits % chunk_bits == 0` check is added in this commit (https://github.com/succinctlabs/succinctx/pull/331/commits/454e91dbea8f1eea5c1578c290a90b3bdc5fd12e).

16. [Informational] Miscellaneous Observations

The tests in `eddsa.rs` in plonky2x's ecc folder is failing - this is due to there being a 16 limb by 16 limb multiplication. As `MAX_NUM_ADDENDS` is 16, the assertion `num_addends <= MAX_NUM_ADDENDS` fails. A potential fix is to set `MAX_NUM_ADDENDS` to `32` or `64` , increasing the carry to be at most 3 limbs instead of 2 limbs. Another way is to make the product to be 8 limb by 16 limb, as the one of the multiplicand is the constant 256 bit scalar modulus. Note that ignoring the `debug_assert` that's failing, the circuit is perfectly fine.

The `select_array<V: CircuitVariable>` function in plonky2x allows `selector >= array.len()` , and in that case the output will be `array[0]` . This should be clarified.

```
// vars/array.rs

/// Given `array` of variables and dynamic `selector`, returns `array[selecto
pub fn select_array<V: CircuitVariable>(&mut self, array: &[V], selector: Var
    // The accumulator holds the variable of the selected result
    let mut accumulator = array[0].clone();

    for i in 0..array.len() {
        // Whether the accumulator should be set to the i-th element (if sele
        // Or should be set to the previous value (if selector_enabled=false)
        let target_i = self.constant::<Variable>(L::Field::from_canonical_usi
        let selector_enabled = self.is_equal(target_i, selector);
        // If selector_enabled, then accum_var gets set to arr_var, otherwise
        accumulator = self.select(selector_enabled, array[i].clone(), accumul
    }

    accumulator
}
```

A similar story happens in `merkle` - even if `nb_enabled_leaves` is very large, the circuit still works fine, and in this case all leaves will simply be enabled. This should also be clarified.

```
// merkle/simple.rs
// Fill in the disabled leaves with empty bytes.
let mut is_enabled = self._true();
for i in 0..padded_nb_leaves {
    let idx = self.constant::<Variable>(L::Field::from_canonical_usize(i));

    // If at_end, then the rest of the leaves (including this one) are disabl
    let at_end = self.is_equal(idx, nb_enabled_leaves);
    let not_at_end = self.not(at_end);
    is_enabled = self.and(not_at_end, is_enabled);

    current_nodes[i] = self.select(is_enabled, current_nodes[i], empty_bytes)
}
```

The `array_contains` function in plonky2x's `vars/array.rs` returns false when there are more than one desired element in the array, which is counterintuitive.

```
pub fn array_contains<V: CircuitVariable>(&mut self, array: &[V], element: V)
    assert!(array.len() < 1 << 16);
    let mut accumulator = self.constant::<Variable>(L::Field::from_canonical_

    for i in 0..array.len() {
        let element_equal = self.is_equal(array[i].clone(), element.clone());
        accumulator = self.add(accumulator, element_equal.variable);
    }

    let one = self.constant::<Variable>(L::Field::from_canonical_usize(1));
    self.is_equal(one, accumulator)
}
```

Fix Notes

The `MAX_NUM_ADDENDS` issue is fixed in this commit (https://github.com/succinctlabs/succinctx/pull/308), increasing the parameter to `64`.

`array_contains` is removed in this pull request (https://github.com/succinctlabs/succinctx/pull/346).

17. [High] `lt` in `ops/math.rs` incorrectly handles zero

```
/// The less than operation (<).
pub fn lt<Lhs, Rhs>(&mut self, lhs: Lhs, rhs: Rhs) -> BoolVariable
where
    Lhs: LessThanOrEqual<L, D, Lhs>,
    Rhs: Sub<L, D, Rhs, Output = Lhs> + One<L, D>,
{
    let one = self.one::<Rhs>();
    let upper_bound = rhs.sub(one, self);
    self.lte(lhs, upper_bound)
}
```

To check if `a < b`, the function checks if `a <= b - 1` - however, it doesn't check for underflows for the case `b = 0`, leading to incorrect proofs such as `0 > 5`.

```
fn test_math_gt() {
    let mut builder = DefaultBuilder::new();

    let v0 = builder.read::<U32Variable>();
    let v1 = builder.read::<U32Variable>();
    let result = builder.read::<BoolVariable>();
    let computed_result = builder.gt(v0, v1);
    builder.assert_is_equal(result, computed_result);

    let circuit = builder.build();

    let test_cases = [
        (10u32, 20u32, false),
        (10u32, 10u32, false),
        (0u32, 5u32, true),
    ];

    for test_case in test_cases.iter() {
        let mut input = circuit.input();
        input.write::<U32Variable>(test_case.0);
        input.write::<U32Variable>(test_case.1);
        input.write::<BoolVariable>(test_case.2);

        let (proof, output) = circuit.prove(&input);
        circuit.verify(&proof, &input, &output);
    }
}
```

This vulnerability was found independently by both auditor and the Succinct team.

Fix Notes

Fixed in this pull request (https://github.com/succinctlabs/succinctx/pull/338), by checking `a < b` by taking the `NOT` of `b <= a` .

18. [High] `sha256` , `sha512` padding is incomplete and potentially underconstrained

In the case of variable length hashing, the padding function needs to constrain that

- the resulting padded input is correct
- the claimed input byte length (which is variable) is sound

The first issue is that the variable length SHA256 is incomplete. In the case where `input.len() = input_byte_length = 64` , the padded input is supposed to have 128 bytes. However, the padding function asserts that the returned result is of length `input.len()` , as the function doesn't account for the increased length of the input due to the padding.

```
pub(crate) fn pad_message_sha256_variable(
    &mut self,
    input: &[ByteVariable],
    input_byte_length: U32Variable,
) -> Vec<ByteVariable> {
    let max_number_of_chunks = input.len() / 64;
    assert_eq!(
        max_number_of_chunks * 64,
        input.len(),
        "input length must be a multiple of 64 bytes"
    );
    // .....
    assert_eq!(padded_bytes.len(), max_number_of_chunks * 64);
    padded_bytes
}
```

This leads to, at minimum, a completeness issue, shown by the following test.

```
#[test]
#[cfg_attr(feature = "ci", ignore)]
fn test_sha256_curta_variable_single_full() { // should pass but fails
    env::set_var("RUST_LOG", "debug");
    env_logger::try_init().unwrap_or_default();
    dotenv::dotenv().ok();

    let mut builder = CircuitBuilder::<L, D>::new();

    let byte_msg : Vec<u8> = bytes!("00de6ad0941095ada2a7996e6a888581928203b8
    let msg = byte_msg
        .iter()
        .map(|b| builder.constant::<ByteVariable>(*b))
        .collect::<Vec<_>>();

    let bytes_length = builder.constant::<U32Variable>(64);

    let expected_digest =
        bytes32!("e453f1f553b165200e0522d448ec148bd67976249fb27513a1c9b264280
    let expected_digest = builder.constant::<Bytes32Variable>(expected_digest

    let msg_hash = builder.curta_sha256_variable(&msg, bytes_length);
    builder.watch(&msg_hash, "msg_hash");
    builder.assert_is_equal(msg_hash, expected_digest);

    let circuit = builder.build();
    let input = circuit.input();
    let (proof, output) = circuit.prove(&input);
    circuit.verify(&proof, &input, &output);

    circuit.test_default_serializers();
}
```

Compare this to the SHA512 padding, which accounts for the increase of length.

```
pub(crate) fn pad_sha512_variable_length(
    &mut self,
    input: &[ByteVariable],
    input_byte_length: U32Variable,
) -> Vec<ByteVariable> {
    let last_chunk = self.compute_sha512_last_chunk(input_byte_length);

    // Calculate the number of chunks needed to store the input. 17 is the nu
    // by the padding and LE length representation.
    let max_num_chunks = ceil_div_usize(input.len() + 17, SHA512_CHUNK_SIZE_E

    // Extend input to size max_num_chunks * 128 before padding.
    let mut padded_input = input.to_vec();
    padded_input.resize(max_num_chunks * SHA512_CHUNK_SIZE_BYTES_128, self.ze
    // .....
}
```

It's also a notable issue in all paddings (including `blake2b`) that there is no direct constraint that `input_byte_length <= input.len()`. This leads to the following observation - in SHA512, if `input.len() = 2` and `input_byte_length = 3`, the padding will be done as if the input is an array of length 3 and hash the padded input accordingly.

This is shown by the following test, evaluating `sha512(0x0102)` with `length = 3`.

```
#[test]
#[cfg_attr(feature = "ci", ignore)]
fn test_sha512_variable_length_weird() {
    setup_logger();
    let mut builder = DefaultBuilder::new();

    let total_message = [1u8, 2u8];
    let message = total_message
        .iter()
        .map(|b| builder.constant::<ByteVariable>(*b))
        .collect::<Vec<_>>();

    let expected_digest = sha512(&[1u8, 2u8, 0u8]);

    let length = builder.constant::<U32Variable>(3 as u32);

    let digest = builder.curta_sha512_variable(&message, length);
    let expected_digest = builder.constant::<BytesVariable<64>>(expected_dige
    builder.assert_is_equal(digest, expected_digest);

    let circuit = builder.build();
    let input = circuit.input();
    let (proof, output) = circuit.prove(&input);
    circuit.verify(&proof, &input, &output);
}

// need to patch digest_hint.rs for this as follows
// this is practically for witness generating purposes
impl<
        L: PlonkParameters<D>,
        H: Hash<L, D, CYCLE_LEN, USE_T_VALUES, DIGEST_LEN>,
        const D: usize,
        const CYCLE_LEN: usize,
        const USE_T_VALUES: bool,
        const DIGEST_LEN: usize,
    > Hint<L, D> for HashDigestHint<H, CYCLE_LEN, USE_T_VALUES, DIGEST_LEN>
{
    fn hint(&self, input_stream: &mut ValueStream<L, D>, output_stream: &mut
        let length = input_stream.read_value::<Variable>().as_canonical_u64()
        // Read the padded chunks from the input stream.
        let length = 2;
        let message = input_stream.read_vec::<ByteVariable>(length);
        let message = Vec::from([1u8, 2u8, 0u8]);

        let digest = H::hash(message);
        // Write the digest to the output stream.
        output_stream.write_value::<[H::IntVariable; DIGEST_LEN]>(digest)
    }
}
```

To prevent this, we recommend adding `input_byte_length <= input.len()` . This is also beneficial in preventing all potential overflows with `input_byte_length` . For example, `input_byte_length * 8` is used to calculate the number of bits in the input - but there are no checks that this is within u32 range. By forcing `input_byte_length <= input.len()` , we avoid such issues. We note that this constraint also clearly removes the possibility that `last_chunk` is larger than the actual number of chunks in the returned padded input.

Fix Notes

Fixed in [this pull request](https://github.com/succinctlabs/succinctx/pull/342) (https://github.com/succinctlabs/succinctx/pull/342). The padding function for SHA256 and SHA512 are now similar as well - both pad the input array correctly, and both enforce `input_byte_length <= input.len()` . The padding circuit for Blake2b also enforces this inequality, added [here](https://github.com/succinctlabs/succinctx/pull/347) (https://github.com/succinctlabs/succinctx/pull/347).

19. [Low] `digest` should be allocated with `init` instead of `init_unsafe` for additional safety

In all hash related curta functions, the `digest` is allocated with `init_unsafe` .

```
pub fn curta_sha256(&mut self, input: &[ByteVariable]) -> Bytes32Variable {
    if self.sha256_accelerator.is_none() {
        self.sha256_accelerator = Some(SHA256Accelerator {
            hash_requests: Vec::new(),
            hash_responses: Vec::new(),
        });
    }

    let digest = self.init_unsafe::<Bytes32Variable>();
    let digest_array = SHA256::digest_to_array(self, digest);
    let accelerator = self
        .sha256_accelerator
        .as_mut()
        .expect("sha256 accelerator should exist");
    accelerator
        .hash_requests
        .push(HashRequest::Fixed(input.to_vec()));
    accelerator.hash_responses.push(digest_array);

    digest
}
```

This is interesting, as `digest_array` is the one that gets passed on to the curta prover. This is, in the case of SHA256, the result of `U32Variable::decode` with `digest` . Since the targets of `digest` are not constrained to be boolean, a potential attack idea is to use overflow to create a `digest` that is not a proper `Bytes32Variable` , but decodes to the correct `digest_array` . This attack is not possible, but due to a rather unexpected reason.

The decode function takes all the targets and runs them to `le_sum`, then returns the `U32Variable` with `from_variables_unsafe` (assuming the result is within range).

```
 fn decode<L: PlonkParameters<D>, const D: usize>(
    builder: &mut CircuitBuilder<L, D>,
    bytes: &[ByteVariable],
) -> Self {
    assert_eq!(bytes.len(), 4);

    // Convert into an array of BoolTargets.  The assert above will guarantee
    // will have a size of 32.
    let mut bits = bytes.iter().flat_map(|byte| byte.targets()).collect_vec()
    bits.reverse();

    // Sum up the BoolTargets into a single target.
    let target = builder
        .api
        .le_sum(bits.into_iter().map(BoolTarget::new_unsafe));

    // Target is composed of 32 bool targets, so it will be within U32Variabl
    Self::from_variables_unsafe(&[Variable(target)])
}
```

So if `le_sum` doesn't perform a boolean range check over the targets, the attack does work.

The `le_sum` works in two ways depending on `num_bits` and the configuration.

If `arithmetic_ops <= self.num_base_arithmetic_ops_per_gate()`, the function directly accumulates the bits without range check, leading to an attack. In the other case, `BaseSumGate<2>` is used, which actually performs a range check over the bits.

```
pub fn le_sum(&mut self, bits: impl Iterator<Item = impl Borrow<BoolTarget>>)
    let bits = bits.map(|b| *b.borrow()).collect_vec();
    let num_bits = bits.len();
    assert!(
        num_bits <= log_floor(F::ORDER, 2),
        "{} bits may overflow the field",
        num_bits
    );
    if num_bits == 0 {
        return self.zero();
    }

    // Check if it's cheaper to just do this with arithmetic operations.
    let arithmetic_ops = num_bits - 1;
    if arithmetic_ops <= self.num_base_arithmetic_ops_per_gate() {
        let two = self.two();
        let mut rev_bits = bits.iter().rev();
        let mut sum = rev_bits.next().unwrap().target;
        for &bit in rev_bits {
            sum = self.mul_add(two, sum, bit.target);
        }
        return sum;
    }

    debug_assert!(
        BaseSumGate::<2>::START_LIMBS + num_bits <= self.config.num_routed_wi
        "Not enough routed wires."
    );
    let gate_type = BaseSumGate::<2>::new_from_config::<F>(&self.config);
    let row = self.add_gate(gate_type, vec![]);
    for (limb, wire) in bits
        .iter()
        .zip(BaseSumGate::<2>::START_LIMBS..BaseSumGate::<2>::START_LIMBS + r
    {
        self.connect(limb.target, Target::wire(row, wire));
    }
    for l in gate_type.limbs().skip(num_bits) {
        self.assert_zero(Target::wire(row, l));
    }

    self.add_simple_generator(BaseSumGenerator::<2> { row, limbs: bits });

    Target::wire(row, BaseSumGate::<2>::WIRE_SUM)
}
```

Therefore, the safety of `digest` depends on the configuration. In our case, `num_bits = 32`. The `num_base_arithmetic_ops_per_gate` can be seen to be at most `config.num_routed_wires / 4` by the code below.

```rust
// circuit_builder
pub(crate) const fn num_base_arithmetic_ops_per_gate(&self) -> usize {
    if self.config.use_base_arithmetic_gate {
        ArithmeticGate::new_from_config(&self.config).num_ops
    } else {
        self.num_ext_arithmetic_ops_per_gate()
    }
}

pub(crate) const fn num_ext_arithmetic_ops_per_gate(&self) -> usize {
    ArithmeticExtensionGate::<D>::new_from_config(&self.config).num_ops
}

// ArithmeticGate
pub const fn new_from_config(config: &CircuitConfig) -> Self {
    Self {
        num_ops: Self::num_ops(config),
    }
}

pub(crate) const fn num_ops(config: &CircuitConfig) -> usize {
    let wires_per_op = 4;
    config.num_routed_wires / wires_per_op
}

// ArithmeticExtensionGate
pub const fn new_from_config(config: &CircuitConfig) -> Self {
        Self {
            num_ops: Self::num_ops(config),
        }
    }

pub(crate) const fn num_ops(config: &CircuitConfig) -> usize {
    let wires_per_op = 4 * D;
    config.num_routed_wires / wires_per_op
}
```

Thankfully, the stadard configuration has `num_routed_wires = 80` .

```
pub const fn standard_recursion_config() -> Self {
    Self {
        num_wires: 135,
        num_routed_wires: 80,
        num_constants: 2,
        use_base_arithmetic_gate: true,
        security_bits: 100,
        num_challenges: 2,
        zero_knowledge: false,
        max_quotient_degree_factor: 8,
        fri_config: FriConfig {
            rate_bits: 3,
            cap_height: 4,
            proof_of_work_bits: 16,
            reduction_strategy: FriReductionStrategy::ConstantArityBits(4, 5)
            num_query_rounds: 28,
        },
    }
}
```

We believe the circuit soundness should not depend on prover configurations, so we recommend changing all `init_unsafe` on hashes to `init` for additional safety.

Fix Notes

Fixed by changing all `init_unsafe` to `init` in this pull request (https://github.com/succinctlabs/succinctx/pull/345).

20. [High] Incorrect Fiat-Shamir and variable length handling in `get_fixed_subarray`

The function `get_fixed_subarray` uses the standard RLC trick to constrain that the correct subarray is being returned. The randomness for the RLC is generated by hashing a provided `seed`, an array of `ByteVariable`. The usual Fiat-Shamir heuristic enforces that this `seed` should contain the entire array as well as the claimed subarray result. However, the definition of `seed` is not documented clearly, and the unit test for this function uses a fixed constant for `seed`, leading to possible confusion on the specifications for `seed`. Indeed, such mistakes were done on the VectorX codebase, such as here (https://github.com/succinctlabs/vectorx/blob/814909d5a8dad0804f40b5935341c19d4de5f643/circuits/builder/decoder.rs#L133), where only the array hash is used.

There is also the usual issues dealing with variable length objects and RLCs - handling trailing zeroes. The function supports variable `start_idx` and fixed `SUB_ARRAY_SIZE`, and works by

- computing `accumulator1`, the RLC of the elements with index within the range `[start_idx, start_idx + SUB_ARRAY_SIZE)` in the original array

- computing `accumulator2` , the RLC of the elements in the claimed subarray
- checking `accumulator1 = accumulator2`

Now there's the potential issue of having `start_idx + SUB_ARRAY_SIZE >= ARRAY_SIZE` - by simply putting zeros at the claimed subarray for indices that are out of bounds in the original array, we can make both accumulators equal even though the actual lengths are different. For example, with array `[0, 1, 2]` , we can show that `array[1..5] = [1, 2, 0, 0]` . It is up to the team to decide whether this is intended by analyzing how the function is used.

Fix Notes

The fix is added in this pull request (https://github.com/succinctlabs/succinctx/pull/352). We briefly explain the fix here.

There are now two functions, `extract_subarray` and `get_fixed_subarray` .

The `extract_subarray` function takes in a `array` , `sub_array` , `start_idx` , `seed` . The `seed` is **expected** to contain both `array` and `sub_array` in some way. This function uses the usual RLC method from challenges derived from `seed` to verify that `sub_array` is indeed the correct subarray from `start_idx` to `start_idx + SUB_ARRAY_SIZE` of `array` . There is also a check that the length of the subarray is precisely `SUB_ARRAY_SIZE` .

The `get_fixed_subarray` function also takes in a `array` , `start_idx` , and a `seed` . The assumption is that `seed` is a verified hash of the `array` or the `array` itself. It first generates the `sub_array` based on the hint generator, then appends it to the `seed` . By calling the `extract_subarray` function, it validates that `sub_array` is a correct subarray of the `array` .

The relevant fix at the VectorX can be seen in this pull request (https://github.com/succinctlabs/vectorx/pull/109).

The variable length handling is fixed in this pull request (https://github.com/succinctlabs/succinctx/pull/346) as recommended.