PUBLIC

Code Assessment

of the Plonky2 BN254 & Keccak256 Circuits

December 20, 2024

Produced for

by

Contents

1 Executive Summary

Dear all,

Thank you for trusting us to help Intmax with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed circuits of Plonky2 BN254 & Keccak256 according to Scope to support you in forming an opinion on their security risks.

Intmax implements two specialized libraries for use in Intmax2 ZKP: Plonky2 BN254 for scalar multiplication on the bn254 curve with additional utility functions and Plonky2 Keccak, a circuit gadget that calculates keccak256 hashes compatible with Solidity.

The most critical subjects covered in our audit are soundness and completeness. Before the intermediate report, several missing constraints allowed proving arbitrary statements:

- Padding Filter Allows Bypassing STARK Constraints
- Missing Constraints for Some starting values of STARKs

For details and further issues, please refer to the detailed issue description in the report.

No issues were uncovered in Plonky2 Keccak.

After the intermediate report all issues have been resolved.

In summary, we find that the Plonky2 Keccak256 and Plonky2 BN254 codebases provide a high level of security.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered, and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	3
• Code Corrected	3
High-Severity Findings	1
• Code Corrected	1
Medium-Severity Findings	0
Low-Severity Findings	1
• Code Corrected	1

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the Plonky2 BN254 & Keccak256 repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

Plonky2 BN254

V	Date	Commit Hash	Note
1	27 August 2024	5cec2397fbaeb5530516ab3eb62510fb98bd9755	Initial Version
2	16 December 2024	7c0fbf3cee1ebf38a85dd4cc9db85b63b07f80f3	After Intermediate Report
3	20 December 2024	750c456ca99404e96c7ced6f90316d3cea6a5ffe	After second Intermediate Report
4	20 December 2024	bf875be8b5a0806d7bb2a2d46fabc17d7f0f934e	Final Changes

Plonky2 Keccak

V	Date	Commit Hash	Note
1	27 August 2024	8bf18d3e66e5c2f2782e9f506e434417d8ebd332	Initial Version

A forked version of Plonky2 at commit `4813d563` is used for the zero knowledge proofs.

The scope of this review is the completeness and soundness of the ZKP circuits and CTL-related functions in the following files:

1. Plonky2 BN254:

```
hook.rs
builder.rs

curves/
    g1.rs
    g2.rs

fields/
    biguint.rs
    fq.rs
    fq2.rs

generators/
    to_u16.rs

    fq/
        mod.rs
```

```
        stark_proof.rs

    g1/
        mod.rs
        stark_proof.rs

    g2/
        mod.rs
        stark_proof.rs

starks/
    utils.rs

    common/
        ctl_values.rs
        eq.rs
        round_flags.rs
        verifier.rs

    curves/
        g1/
            add.rs
            mod.rs
            scalar_mul_ctl.rs
            scalar_mul_stark.rs
            scalar_mul_view.rs

        g2/
            add.rs
            mod.rs
            scalar_mul_ctl.rs
            scalar_mul_stark.rs
            scalar_mul_view.rs

            ext/
                add.rs
                convert.rs
                is_modulus_zero.rs
                mod.rs
                modulus_zero.rs
                mul.rs
                sub.rs

        fields/
            exp_ctl.rs
            exp_stark.rs
            exp_view.rs
            mul.rs

        modular/
            is_modulus_zero.rs
            modulus_zero.rs
            pol_utils.rs

utils/
```

```
    g1_msm.rs
    hash_to_g2.rs
```

2. Plonky2 Keccak:

```
builder.rs
circuit_utils.rs
hook.rs

generators/
    stark_proof_generator.rs

keccak_stark/
    columns.rs
    constants.rs
    ctl_values.rs
    keccak_stark.rs
    logic.rs
    prover.rs::keccak_ctls()
    round_flags.rs
    verifier.rs
```

This review focused on the libraries as used in Intmax2-zkp. Caution is advised when these libraries are used in other projects. Reliability and/or the correct usage in other projects is not covered by this review.

2.1.1 Excluded from scope

- Plonky2 soundness and correctness properties
- Plonky2 implementation
- Plonky2 functions and libraries, including `starky` and `plonky_u32`
- Witness generation / trace computation / proof generation
- Tests
- For Plonky2 Keccak: `prover.rs` except the `keccak_ctls()` function
- All the files not explicitly listed above
- Compilation of the circuits with features other than the empty `default = []` is out of the scope of this review

2.2 System Overview

This system overview describes the initially received version (Version 1) as defined in the Assessment Overview.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

Intmax provides two specialized libraries for use in Intmax2: Plonky2 BN254 and Plonky2 Keccak.

2.2.1 Plonky2 BN254

Intmax offers a Plonky2 library with implementation of the base field (`Fq`), quadratic extension of the base field (`Fq2`), `G1` and `G2` groups for the optimal Ate pairing of the BN254 elliptic curve, as well as gadgets for exponentiation in `Fq` and scalar multiplication in `G1` and `G2` implemented as STARKs with `starky`. The library also exposes helper functions for multi scalar multiplication in `G1` and hashing of (Goldilocks) field elements into `G2`. The STARKs are also implemented so they can be recursively verified in a Plonky2 circuit. The STARKs are implemented with `starky`.

A description of the BN254 curve and the groups used for the pairing can be found here: https://web.archive.org/web/20241005042437/https://hackmd.io/@jpw/bn254.

A reference to hashing to elliptic curves can be found here: https://web.archive.org/web/20240929224352/https://datatracker.ietf.org/doc/rfc9380/.

2.2.1.1 BN254 base field and pairing groups (Plonky2)

The base field `Fq` for BN254 has characteristic `q=21888242871839275222246405745257275088696311157297823662689037894645226208583`. The quadratic extension field `Fq2` is defined as `Fq\(u^2+1)` and is needed as `G2` lives in the sextic twist of the curve. Both fields implement functions to constrain basic operations such as:

- addition
- negation
- multiplication
- multiplicative inverse
- sign (as defined by the `sgn0` function in https://web.archive.org/web/20240929224352/https://datatracker.ietf.org/doc/rfc9380/)
- equality check
- modulo reduction
- square root with sign
- Legendre symbol (the exponentiation is carried out in the STARK described below)

The group `G1` is defined as the points of the curve $y^2 = x^3 + 3$, with $x, y \in Fq$ and order `r = 21888242871839275222246405745257275088548364400416034343698204186575808495617`. The group `G2` is defined as a subgroup of order `r` of the points of the twisted curve $y^2 = x^3 + \frac{3}{9+u}$, with $x, y \in Fq2$. Both groups implement functions to constrain basic operations such as:

- addition
- negation
- "on curve" check

The implementation of `G1` has two additional functions:

- recoverability check of a point from its `x` coordinate
- recoverability of a point from its `x` coordinate (assuming the recoverability check passed). This is always done assuming the sign (`sgn0`) of the `y` coordinate being even.

2.2.1.2 STARKs (starky)

The STARKs implemented by Intmax are:

- exponentiation in `Fq` (`exp`) implemented with fast-exp (square-and-multiply): `x^s`
- scalar multiplication in `G1` (`g1/scalar_mul`) implemented with double-and-add: `s*P1`

- scalar multiplication in G2 (`g2/scalar_mul`) implemented with double-and-add: `s*P2`

They are implemented as a builder hook that will accumulate inputs and outputs. The maximum size for s is 256bits, this means that the maximum number of steps for a computation to be carried is 512 with square-and-multiply or double-and-add. Therefore, each computation will have 512 rows. After a computation is done, either another one starts on the next row, or padding rows are added. This gives a sense of periodicity in the trace. For the three operations listed above, the period is 512.

The underlying logic of those algorithms being similar, the STARK columns are also similar. They are described below:

- square (`exp`) / double (`*/scalar_mul`): current base for multiplication / addition
- product (`exp`) / sum (`*/scalar_mul`): running product / running sum
- a: holds either square/double or product/sum depending on whether the current row is `is_mul` or `is_sq_not_last`
- b: holds the current base (square/double of the previous row)
- c: holds the result of `a * b / a + b`
- mul_aux (`exp`) / add_aux (`*/scalar_mul`): helper column to constrain the result of c
- bits: `s` bit-representation little endian. This is rotated to the left every time a square/doubling happens, i.e., every two row
- round_flags: flags indicating the first and last rows of a period
- timestamp: timestamp of the computation being carried
- is_mul (`exp`) / is_add (`*/scalar_mul`): flag indicating whether a row is a multiplication/addition row
- is_sq_not_last (`exp`) / is_doubling_not_last (`*/scalar_mul`): flag indicating whether a row is a squaring/doubling row, and that is it not the last operation of the period
- filter: indicates whether the row is part of a computation or padding
- frequency: frequency of every range-checked values, will be used in the range check protocol implemented as logUp in `starky`
- range_counter: list of the values for range check. In our case `[0, 2^16-1]`.

2.2.1.3 Utils

Furthermore, the following functions leveraging scalar multiplication STARKs have been implemented specifically tailored for use in intmax2-zkp:

- `g1_msm()`: performs multi scalar multiplication on the array of inputs.
- `hash_to_g2_circuit()`: Part of the `HashToG2` trait, this function hashes Goldilocks field elements into `G2`. It leverages `map_to_g2_circuit()` also of `HashToG2` to constrain the mapping of points in `Fq2` to `G2` according to the Shallue-van de Woestijne method (6.6.1. Shallue-van de Woestijne Method from https://datatracker.ietf.org/doc/rfc9380/).

2.2.2 Plonky2 Keccak

Intmax offers a Plonky2 Keccak256 gadget implemented as a builder hook that will accumulate inputs and outputs. For each instance of Keccak256, the padding of the message and management of the permutations' input state are checked with Plonky2. The permutation itself, `keccak-f`, is a STARK implemented with `starky`.

2.2.2.1 Keccak without permutation circuit (Plonky2)

For each message to be hashed, the circuit first constrains the message to be padded to a length multiple of the rate (1088), it defines targets to represent the permutation's input and output for each block. The circuit also forces the starting state to be r=the first 1088 bits of the message and c=512 zeros, and then the input of each subsequent permutation to be the output of the previous permutation XORed with the next 1088 bits of the padded message. The final output of Keccak256 will be the first 256bits (8x 32bits limbs) of the unique squeeze round.

The inputs and outputs for each permutation of each message to be hashed is saved in two array of targets `perm_inputs` and `perm_outputs`, where an element of the arrays is a state represented by 5x5 2x(32bits limbs) to make up for the 5x5 64 bits lanes described in the Keccak specification (https://csrc.nist.gov/pubs/fips/202/final).

In summary, this circuit constrains the padding of the messages, the initialization and update of the state between two permutations, and the output to be correctly taken from the state.

2.2.2.2 keccak-f permutation constraints (starky)

The correctness of the `keccak-f` permutation is enforced by a STARK putting constraints on the row of an execution trace. The columns of the STARK as well as the constraints are described in Polygon's zkEVM specification https://0xpolygonzero.github.io/zk_evm/tables/keccak.html. The constraints allow multiple traces of Keccak256 to be concatenated.

2.2.2.3 Keccak256 STARK proof generator circuit (Plonky2)

This circuit brings together the previous two chapters by verifying the STARK of the keccak-f permutations where the columns 25-74 of the first row are the input and columns 2429, 2430, and 2314-2364 of the 24th row are the output for each table. It then links those columns to the targets in `perm_inputs` and `perm_outputs` with the cross-table lookup protocol described in the zkEVM specification. The table that is being proved is the concatenation of the execution trace of each Keccak256 instance.

In summary, this circuit enforces the correctness of a permutation `perm_inputs->perm_outputs` by connecting the elements of `perm_inputs/perm_outputs` to the "input"/"output" parts of the valid STARK tables.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Open Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Design: Architectural shortcomings and design inefficiencies
- Soundness: The verifier can be convinced of a false statement

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	0

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Open Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	3

- Missing Constraints for Some Starting Values of STARKs Code Corrected
- Padding Filter Allows Bypassing STARK Constraints Code Corrected
- Take Modulo Can Be Tricked Code Corrected

High-Severity Findings	1

- Limbs Are Too Long for Library Code Corrected

Medium-Severity Findings	0

Low-Severity Findings	1

- Wrong Comments Code Corrected

Informational Findings	1

- Big Integer Substraction Can Wrap Around Code Corrected

6.1 Missing Constraints for Some Starting Values of STARKs

Design Soundness Critical Version 1 Code Corrected

CS-IBK-001

In the BN254' STARKs for exponentiation and multiscalar multiplication, the columns `square` / `double`, `product` / `sum` of the `exp` and both `scalar_mul` are not constrained to be equal to `b` and `b` / `1` respectively at the beginning of the computation, i.e., when `is_first_round` is true.

This allows to set arbitrary values in the first round of each computation and thus generate proofs for incorrect results.

Code corrected:

When `is_first_round` is true, the values in `square`/`double` and `product`/`sum` columns are now constrained to match the expected starting values derived from other columns:

Exp:

The column `square` represents the current base for the multiplication. In the first round this is now constrained to equal to `b`, which holds the current base.

Column `product` of the first rounds is constrained to `c` if the LSB of the exponent is `1`, or to `a` (separately constrained to `1` in the first rounds) otherwise. `c` is separately constrained as the correct result of the multiplication.

Scalar_mul:

The column `double` represents the current base for the addition. In the first round this is now constrained to equal to b, which holds the current base.

Column `sum` of the first rounds is constrained to c if the LSB of the scalar is `1`, or to a (separately constrained to be the offset in the first rounds) otherwise. c is separately constrained to be the correct result of the addition.

6.2 Padding Filter Allows Bypassing STARK Constraints

Soundness **Critical** **Version 1** **Code Corrected**

CS-IBK-002

In the STARKs of plonky2_bn254 the `filter` column is intended to disable the STARK constraints only for padding rows. However, setting it to 0 across all rows allows to disable the STARK constraints entirely. Since the `input_filter` and `output_filter` columns are not connected to the `filter` column, the CTL lookups still work. The values can be set arbitrarily as needed, enabling any statement to be proven.

Example:

We can prove $2^2 = 42$ using four rows, the remaining rows are padding rows.

- Row 1: generate first row as usual from `x = 2` and `s = 2`, but set `filter = F::ZERO`
- Row 2: generate first transition row as usual, but set `filter = F::ZERO` and `is_sq_not_last = F::ZERO`
- Row 3: padding row
- Row 4: generate the last transition row (`round_flags.is_last_round` will be `F::ONE` for CTL) based on some random starting state, but set `filter = F::ZERO`, `is_sq_not_last = F::ZERO`, and `product` to 42, that will be the output of our computation.

Overall, the relationships between the `filter`, `is_mul`, `is_sq_not_last` and the CTLs filter columns can be strengthened.

Code corrected:

The constraints of the round flags have been updated to enforce the following:

- if `filter` is disabled, then `is_first_round` must be disabled
- if `filter` is disabled, then `is_last_round` must be disabled
- if `is_first_round` is enabled, then `filter` must be enabled
- if `is_last_round` is enabled, then `filter` must be enabled

This ensures that the columns read in the CTL are part of a checked computation because:

- `is_first_round/is_last_round` cannot be set without an active filter,
- when `is_first_round` is enabled, the `counter` must be `0`. `is_first_round` must be disabled otherwise,
- when `is_last_round` is enabled, the `counter_prime` must be `0`. `is_last_round` must be disabled otherwise,
- as soon as `filter` and `is_first_round` are enabled, the following table must contain a valid computation with a `filter` enabled over the 512 next rows

6.3 Take Modulo Can Be Tricked

Soundness Critical Version 1 Code Corrected

CS-IBK-003

The function `Fq::take_mod` of plonky2_bn254 can return a wrong result when the value passed is a multiple of the modulus, this is because the function `div_rem_biguint()` will put constraints that accept a remainder smaller or equal to the divisor. As a result, if the value is a multiple of the divisor it is possible to set the result as the modulus value and set the flag `mod_taken` to true.

Example:

It is possible to prove `45 = 8*5 + 5` with `div_rem_biguint()`.

Code corrected:

The function `div_rem_biguint()` has been updated to enforce that the remainder is strictly smaller than the divisor.

6.4 Limbs Are Too Long for Library

Design High Version 1 Code Corrected

CS-IBK-004

The repeated operations done in `map_to_g2_circuit()` without taking the modulus increase the length of the limbs to the point where the limbs become too long. This behavior triggers the `num_addends <= MAX_NUM_ADDENDS` debug assertion in the `plonky2-u32` library in non-release builds.

Code corrected:

The modulus is taken after each `add`, `sub`, and `mul` operation in `map_to_g2_circuit()`.

6.5 Wrong Comments

Design Low Version 1 Code Corrected

CS-IBK-006

Some comments in the codebase of plonky_bn254 do not represent the implementation. Such misleading comments can hurt the security of the system as the reader may build wrong assumptions. Here is a non-exhaustive list:

1. `starks/fields/mul.rs`: the comment over the function `eval_fq_mul()` mentions some elliptic curve arithmetic but the function is about finite field arithmetic
2. `fields/fq2.rs`: the comment over the function `new_unchecked()` should be over `new_checked()`
3. `starks/fields/exp_stark.rs`: the comments over the functions `generate_one_set()` and `generate_first_row()` mention scalar multiplication but the functions are about exponentiation
4. `starks/fields/exp_stark.rs`: the comment "`// first round should be adding`" in the function `eval_ext_circuit()` should mention multiplicating instead of adding

Version 2

1. The comment of the point 3. above has been updated to mention field exponentiation with an offset, but the field exponentiation does not have an offset

Version 3

1. The functions `add()`, `sub()`, `mul()` in `fq2.rs` have been updated to take the modulus, but the comments over the functions still mention they do not take the modulus

---

Code corrected:

1. Fixed: Comment has been corrected.
2. Fixed: Comment has been moved.
3. Fixed in Version 3: Comment has been corrected.
4. Fixed: Comment has been corrected.

Version 2

1. Fixed: Comment has been corrected.

Version 3

1. Fixed: Comments have been corrected.

6.6 Big Integer Substraction Can Wrap Around

Informational | Version 1 | Code Corrected

CS-IBK-007

The function `CircuitBuilderBiguint::sub_biguint` (a-b) does not enforce the last borrow to be zero. If a < b, the result will wrap around on the available limbs, so the caller must ensure a >= b to avoid this behavior.

While this assumption is noted in the interface, the function implementation itself doesn't reiterate the warning. Callers of this function must be fully aware of this assumption, as incorrectly using a < b can lead to critical issues.

Code corrected:

After the intermediate report the code has been changed to include an assert preventing a non-zero borrow at the end:

```
// Borrow should be zero here.
self.assert_zero(borrow.0);
```

7 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

7.1 Aggregated STARK Table Can Be Rotated

Note Version 1

Considering only the BN254 STARKs, it is possible to prove a "rotated" version of the trace table, where the generated trace (before the range check columns are filled) is shifted upwards or downwards. For example, instead of starting at round 0, the table might begin at round 4, with the earlier rounds wrapping around to the bottom.

While this does not pose any security issue when used within Intmax2, users should be aware of this special behavior.

7.2 No Subgroup Check for `G2`

Note Version 1

CS-IBK-005

The implementation of `G2` only enforces the points (x, y) to have $x, y \in F_{q^2}$ and that the point belongs to the curve. But no constraints enforce that the point actually belongs to the subgroup G_2.

From Intmax2-zkp, G2 is used exclusively in `hash_to_g2()`. The pairing check for the hash is done in the smart contract.

This approach is appropriate for the intended use case within Intmax2-zkp. However, if the library is used differently, performing a subgroup check may become necessary.