Axiom Halo2 Libraries

Security Assessment

June 16, 2023

Prepared for:
Yi Sun
Haichen Shen

Prepared by: **Filipe Casal, Jim Miller, Fredrik Dahlgren**
Joe Doyle, Tjaden Hess, and Marc Ilunga

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow @trailofbits on Twitter and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact, or email us at info@trailofbits.com.

Trail of Bits, Inc.
228 Park Ave S #80688
New York, NY 10003
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

The sole canonical source for Trail of Bits publications is the Trail of Bits Publications page. Reports accessed through sources other than that page may have been modified and should not be considered authentic.

Test Coverage Disclaimer

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test the controls and security properties of software. These techniques augment our manual security review work, but each has its limitations: for example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. Their use is also limited by the time and resource constraints of a project.

Table of Contents

Executive Summary

Engagement Overview

Axiom engaged Trail of Bits to review the security of its Halo2 libraries. These libraries implement zero-knowledge circuits using the `halo2` proving system for low-level operations such as inner products, field and elliptic curve arithmetic, and hash functions, as well as Axiom's business logic.

A team of six consultants conducted the review from February 10 to May 17, 2023, for a total of 14 engineer-weeks of effort. Our testing efforts focused on circuit soundness and completeness. With full access to the source code and documentation, we performed static and dynamic testing of the codebase, using automated and manual processes. The `snark_verifier`'s IPA code was out of scope for this audit.

The codebase represents highly complex cutting-edge technology. In particular, the project contains many different components developed using the `halo2` library, each with its own complex logic. We used a combination of manual and automated review to assess all of the components in scope, through which we uncovered three high-severity issues and many other security issues. Due to the high level of complexity of the codebase and the number of security issues we uncovered, we recommend that Axiom consider performing an additional (internal or external) review of the codebase.

Observations and Impact

During the audit, we uncovered three high-severity issues related to underconstrained circuits that would severely compromise the system's security. The codebase suffers from a lack of testing that would have prevented several other issues found during the security assessment. Other common sources of issues in Axiom's codebase, which are common in Rust codebases, include unchecked uses of the `zip` operator, which can lead to missing or incomplete validations, and arithmetic issues, such as overflows and bit-shifts larger than the integer type that lead to runtime errors or incorrect results.

Recommendations

Based on the codebase maturity evaluation and findings identified during the security review, Trail of Bits recommends that Axiom take the following steps:

- **Remediate the findings disclosed in this report.** The findings described in Detailed Findings pose a high risk to the soundness of Halo2 circuits. These findings should be addressed as part of direct remediation or as part of any refactoring that may occur when addressing other recommendations.

- **Invest in testing.** The codebase severely lacks tests. Basic unit tests and negative tests would have prevented several issues that were found during the security assessment. In a rapidly changing codebase such as Axiom's, tests can also prevent potential regressions during code refactoring.

- **Invest in code documentation.** The codebase severely lacks code documentation. We recommend writing documentation for every function, describing the implemented functionality and the preconditions that the function arguments must satisfy. Additionally, higher-complexity code should include algorithm descriptions in plaintext and, if possible, references to external documentation (e.g., the page number of a particular paper with a URL to that paper). This documentation would allow auditors and new developers to quickly compare the specification with the implementation.

- **Consider performing an additional code review.** Given the complexity of the codebase and the number of security issues uncovered during the security assessment, we recommend that Axiom perform an additional code review of the codebase, either internally or externally.

The following tables provide the number of findings by severity and category.

EXPOSURE ANALYSIS

Severity	*Count*
High	3
Medium	6
Low	7
Informational	12
Undetermined	1

CATEGORY BREAKDOWN

Category	*Count*
Cryptography	5
Data Validation	23
Undefined Behavior	1

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager
dan@trailofbits.com

Brooke Langhorne, Project Manager
brooke.langhorne@trailofbits.com

The following engineers were associated with this project:

Filipe Casal, Consultant
filipe.casal@trailofbits.com

Joe Doyle, Consultant
joseph.doyle@trailofbits.com

Jim Miller, Consultant
james.miller@trailofbits.com

Tjaden Hess, Consultant
tjaden.hess@trailofbits.com

Fredrik Dahlgren, Consultant
fredrik.dahlgren@trailofbits.com

Marc Ilunga, Consultant
marc.ilunga@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
January 31, 2023	Pre-project kickoff call
February 10, 2023	Status update meeting #1
February 17, 2023	Status update meeting #2
February 24, 2023	Status update meeting #3
March 13, 2023	Status update meeting #4
March 17, 2023	Status update meeting #5
March 24, 2023	Status update meeting #6
May 17, 2023	Delivery of report draft
May 17, 2023	Report readout meeting
June 16, 2023	Delivery of final report with fix review

Project Goals

The engagement was scoped to provide a security assessment of Axiom's Halo2 libraries. Specifically, we sought to answer the following non-exhaustive list of questions:

- Are the circuits sound, complete, and properly constrained?
- Are cryptographic primitives implemented and used correctly?
- Is field and elliptic curve arithmetic correctly implemented? Are edge-case values (e.g., the point at infinity) handled correctly?
- Is the `halo2` API used in a correct and safe manner?
- Are the developed APIs prone to potential misuse?
- Can assumptions on function inputs be violated in practice?
- Does the codebase follow good Rust programming practices?
- Does the codebase rely on any outdated or insecure dependencies?

Project Targets

The engagement involved a review and testing of the following targets.

halo2-base, zkevm-keccak

Repository	github.com/axiom-crypto/halo2-lib/tree/upgrade-v0.3.0
Version	`eff200f78b88610919c4e3d0accc319f688d98a5`
Types	Rust, Halo2
Platform	Native

halo2-ecc

Repository	github.com/axiom-crypto/halo2-lib/tree/upgrade-v0.3.0
Version	`c31a30bcaff384b0c3aa7c823dd343f5c85da69e`
Types	Rust, Halo2
Platform	Native

snark-verifier

Repository	github.com/axiom-crypto/snark-verifier/tree/sync/halo2-lib-v0.3.0
Version	`efec13b776f5ae84220941b04ef95ae35ffb749d`
Types	Rust, Halo2
Platform	Native

axiom-eth

Repository	github.com/axiom-crypto/axiom-core-working/tree/audit-snapshot
Version	`cfc1116211a44d9b5e700e3ee322f906e83b8090`
Types	Rust, Halo2
Platform	Native

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches and their results include the following:

- `halo2-base` and `halo2-ecc`: We performed a detailed manual review of the circuits in these codebases, focusing on circuit soundness and completeness, as well as functional correctness and proper constraining. Other than reviewing the circuit logic, we looked for general programming issues that could propagate into circuit generation or cause runtime errors.

- `snark-verifier`: We manually reviewed the `snark-verifier` codebase, including the emitted Yul code, focusing on verifier validations. We focused on identifying general logic issues and common issues that we uncovered in similar codebases, and on checking whether issues that we uncovered in `halo2-ecc` could affect `snark-verifier`. The areas that received the most attention were the `verify` and `decide`/`decide_all` functions.

- `zkevm-keccak`: We manually reviewed the `keccak` implementation, following the `keccak` specification and focusing on potential edge cases.

- `axiom-eth`: We manually reviewed the `axiom-eth` implementation, focusing on circuit completeness and soundness, proper constraining, and general correctness. Where necessary, we consulted relevant references and standards, such as the Ethereum RLP documentation.

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. The following list outlines the coverage limitations of the engagement and indicates system elements that may warrant further review:

- `snark-verifier`: Although we performed a detailed manual review of this codebase, it is complex and critical, which means it warrants another (possibly internal) review. Additionally, we did not review the `snark-verifier`'s IPA code, as it was out of scope.

Automated Testing

Trail of Bits uses automated techniques to extensively test the security properties of software. We use both open-source static analysis and fuzzing utilities, along with tools developed in-house, to perform automated testing of source code and compiled software.

Test Harness Configuration

We used the following tools in the automated testing phase of this project:

Tool	Description	Policy
Semgrep	An open-source static analysis tool for finding bugs and enforcing code standards when editing or committing code and during build time	N/A
`cargo-audit`	An open-source static analysis tool used to audit `Cargo.lock` files for crates with security vulnerabilities reported to the RustSec Advisory Database	Appendix C
Clippy	An open-source Rust linter used to catch common mistakes and unidiomatic Rust code	Appendix C

Areas of Focus

Our automated testing and verification focused on the following:

- Identification of general code quality issues and unidiomatic code patterns
- Identification of known vulnerable dependencies

Test Results

The results of this focused testing are detailed below.

halo2-base, halo2-ecc, zkevm-keccak, and axiom-eth

Property	Tool	Result
The project does not import vulnerable dependencies.	`cargo-audit`	**Passed**
The project adheres to Rust best practices by fixing code quality issues reported by linters like Clippy.	Clippy	**Passed**

snark-verifier

Property	Tool	Result
The project does not import vulnerable dependencies.	`cargo-audit`	**Passed**
The project adheres to Rust best practices by fixing code quality issues reported by linters like Clippy.	Clippy	**Code Quality Issues Found**

Codebase Maturity Evaluation

Trail of Bits uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

Category	Summary	Result
Arithmetic	We found three low-severity issues related to overflowing integers that would cause incorrect results (TOB-AXIOM-1, TOB-AXIOM-5, and TOB-AXIOM-20).	**Moderate**
Complexity Management	The codebase is generally well structured. However, we identified an opportunity to use Rust types to enforce data validity requirements at compile-time (TOB-AXIOM-12). This would make the code more readable and ensure that validations are not forgotten. We also found that the `axiom-eth` codebase is very complex, there is little file organization, and functions are very long. The `zkevm-keccak` codebase uses several byte transformations that are generally hard to follow and could warrant more documentation. The purpose and constraints associated with all `halo2` selectors should also be made explicit in the documentation.	**Satisfactory**
Cryptography and Key Management	We found several high- and medium-severity issues caused by underconstrained circuits that could compromise the system's security if exploited (TOB-AXIOM-2, TOB-AXIOM-3, TOB-AXIOM-4, TOB-AXIOM-10, TOB-AXIOM-13, and TOB-AXIOM-19). We also found issues related to missing validations on field and elliptic curve operations (TOB-AXIOM-11, TOB-AXIOM-12, TOB-AXIOM-15, and TOB-AXIOM-18).	**Weak**
Data Handling	We found medium- and low-severity issues related to missing or insufficient input validation, namely insufficient validation of arguments to the `zip` iterator, (TOB-AXIOM-16, TOB-AXIOM-17 and TOB-AXIOM-25) and missing empty array validations (TOB-AXIOM-21).	**Moderate**

Documentation	The codebase severely lacks documentation, both documentation in function headers and inline documentation to describe higher complexity code. We recommend that the Axiom team carefully document each function's intended use and parameters, as well as preconditions that those parameters must satisfy (particularly on public API functions). Furthermore, we recommend explicitly documenting higher-complexity code such as cryptographic primitives with their external references (e.g., URLs to papers and page numbers). This will make the project easier to audit because it would bind algorithm specifications to their implementations.	Weak
Memory Safety and Error Handling	The codebase contains no unsafe code. However, some important circuit validity properties are checked with debug-only assertions, such as length requirements as described in finding TOB-AXIOM-2, or are noted in only comments, such as the elliptic curve order restrictions as described in finding TOB-AXIOM-23.	Satisfactory
Testing and Verification	The codebase severely lacks both unit tests and negative tests. These types of tests could have prevented several findings, including the high-severity finding TOB-AXIOM-19. Tests serve other purposes beyond determining function correctness: they can prevent regressions during code refactorings and allow external auditors and developers to quickly interact with the codebase and understand the functions' intended uses.	Missing

Summary of Findings

The table below summarizes the findings of the review, including type and severity details.

ID	Title	Type	Severity
1	Incorrect limb decomposition due to bit-shifts larger than integer size	Undefined Behavior	Low
2	Risk of unconstrained inner product in release builds	Data Validation	Medium
3	idx_to_indicator circuit is underconstrained	Cryptography	High
4	ecdsa_verify_no_pubkey_check can fail on signatures from crafted public keys	Data Validation	Medium
5	log2_ceil function miscomputes its result when x input is zero	Data Validation	Low
6	GateChip::num_to_bits depends on implementation-specific details of the underlying field	Data Validation	Low
7	RangeChip::get_last_bit returns the wrong value	Data Validation	Low
8	Validations missing in release builds	Data Validation	Medium
9	Keccak implementation cannot hash arbitrarily large inputs	Data Validation	Informational
10	Field division of zero by zero is unconstrained	Data Validation	Medium
11	Incorrect point-at-infinity handling in elliptic curve operations	Cryptography	Medium

12	FpChip::load_private allows non-reduced field elements	Data Validation	**Informational**
13	scalar_multiply can return underconstrained results	Cryptography	**High**
14	Witness may be underconstrained if two gates overlap with more than one cell	Data Validation	**Informational**
15	EccChip::load_private does not enforce that witness values are on-curve	Data Validation	**Informational**
16	Native KZG accumulation decider accepts an empty vector	Cryptography	**Medium**
17	Polynomial addition and subtraction assume polynomials have the same degree	Data Validation	**Informational**
18	FpChip::enforce_less_than_p incorrectly allows certain values above 2^t	Data Validation	**Informational**
19	FpChip::assert_equal does not assert equality	Data Validation	**High**
20	Scalar rotation misbehaves on i32::MIN	Data Validation	**Low**
21	Several functions assume that arguments are non-empty	Data Validation	**Low**
22	EVM verifier does not validate the deployment code	Data Validation	**Informational**
23	Values from load_random_point are used without strict checks	Data Validation	**Informational**
24	query_cell_at_pos assumes that the column index is valid	Data Validation	**Informational**

25	Unchecked uses of zip could bypass checks on parse_account_proof_phase0 and parse_storage_proof_phase0	Data Validation	**Undetermined**
26	The hex_prefix_encode and hex_prefix_encode_first functions assume that the is_odd parameter is a bit	Data Validation	**Informational**
27	batch_invert_and_mul ignores zero elements and panics on empty arrays	Data Validation	**Low**
28	Proof caching occurs before proof validation	Data Validation	**Informational**
29	Merkle root computation does not differentiate leaf data hashing and inner node hashing	Cryptography	**Informational**

Detailed Findings

1. Incorrect limb decomposition due to bit-shifts larger than integer size	
Severity: **Low**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-AXIOM-1
Target: `halo2-base/src/utils.rs`	

Description

The `decompose_u64_digits_to_limbs` and `decompose_biguint` functions incorrectly compute their results: the `decompose_u64_digits_to_limbs` function always returns a zero-filled vector when the target limb size is 64, and the `decompose_biguint` function incorrectly computes the limb values when `bit_len` is larger than 96.

The `decompose_u64_digits_to_limbs` function splits the input `u64` integers into smaller `bit_len`-sized limbs. To do so, it computes a bitmask that it uses to bit-and the input integers, as shown in figure 1.1.

```
pub(crate) fn decompose_u64_digits_to_limbs(
    e: impl IntoIterator<Item = u64>,
    number_of_limbs: usize,
    bit_len: usize,
) -> Vec<u64> {
    debug_assert!(bit_len <= 64);

    let mut e = e.into_iter();
    let mask: u64 = (1u64 << bit_len) - 1u64;
```

Figure 1.1: `halo2-base/src/utils.rs#L63-L72`

However, when `bit_len` equals 64, the mask calculation performs a left-shift equal to the integer size (64), which is undefined behavior. In Rust, this undefined behavior manifests as an overflow panic in debug mode; in release mode, the left-shift silently results in a zero. Thus, when the mask is zero, the function will return a vector with a `number_of_limbs` number of zeroes regardless of the argument integers.

The `decompose_biguint` function uses the `BigUint::iter_u64_digits()` function to obtain the big integer in a 64-bit limb representation. Then, it uses a `u128` variable to represent each limb. For example, $2^{64} + 1$ is represented by two 64-bit limbs, `[1, 1]`; if we want to represent it with two limbs of length 96, the code would use the first 96-bit limb as the first 64-bit limb and then compute the remaining number of bits that fit: 96 - 64 = 32.

Thus, the code would fetch the next 64-bit limb and get 32 bits from it. Figure 1.2 shows the code that implements this logic.

```
pub fn decompose_biguint<F: PrimeField>(e: &BigUint, num_limbs: usize, bit_len:
usize) -> Vec<F> {
    debug_assert!(bit_len > 64 && bit_len <= 128);
    let mut e = e.iter_u64_digits();

    let mut limb0 = e.next().unwrap_or(0) as u128;
    let mut rem = bit_len - 64;
    let mut u64_digit = e.next().unwrap_or(0);
    limb0 |= ((u64_digit & ((1 << rem) - 1)) as u128) << 64;
    u64_digit >>= rem;
    rem = 64 - rem;

    core::iter::once(F::from_u128(limb0))
```

Figure 1.2: `halo2-lib/halo2-base/src/utils.rs#L195-L204`

However, these operations perform bit-shifts larger than the integer size when `rem` is larger than or equal to 32. If `bit_len` is 96, the `((u64_digit & ((1 << rem) - 1)) as u128) << 64` expression would equal `0`, and the remaining `u64_digit` would equal 1. This will cause the big integer decomposition to be incorrectly computed.

The following two test cases test the two affected functions:

```
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_decompose_biguint() {
        use halo2_proofs_axiom::halo2curves::bn256::Fr;
        use num_traits::FromPrimitive;

        let e = BigUint::from_u64(2).unwrap().pow(64) +
BigUint::from_u64(1).unwrap();
        let v = decompose_biguint::<Fr>(&e, 4, 128);
        assert_eq!(
            v,
            vec![Fr::from_u128(1u128 + (1u128 << 64)), Fr::from_u128(0),
Fr::from_u128(0), Fr::from_u128(0)]
        );
    }
// running 1 test
// thread 'utils::tests::test_decompose_biguint' panicked at 'assertion failed:
`(left == right)`
//   left: `[0x0000000000000000000000000000000000000000000000000000000000000001,
0x0000000000000000000000000000000000000000000000000000000000000001,
0x0000000000000000000000000000000000000000000000000000000000000000,
0x0000000000000000000000000000000000000000000000000000000000000000]`,
```

```
//  right: `[0x0000000000000000000000000000000000000000000000010000000000000001,
0x0000000000000000000000000000000000000000000000000000000000000000,
0x0000000000000000000000000000000000000000000000000000000000000000,
0x0000000000000000000000000000000000000000000000000000000000000000]`',
halo2-base/src/utils.rs:351:9
// note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
// test utils::tests::test_decompose_biguint ... FAILED


    #[test]
    fn test_decompose_u64_digits_to_limbs() {
        let e = vec![0x123456789abcdef0, 0x123456789abcdef0, 0x123456789abcdef0];
        let limbs = decompose_u64_digits_to_limbs(e, 4, 64);

        assert_eq!(limbs, vec![0x123456789abcdef0, 0x123456789abcdef0,
0x123456789abcdef0, 0]);
    }
//     running 1 test
// thread 'utils::tests::test_decompose_u64_digits_to_limbs' panicked at 'assertion
failed: `(left == right)`
//   left: `[0, 0, 0, 0]`,
//  right: `[1311768467463790320, 1311768467463790320, 1311768467463790320, 0]`',
halo2-base/src/utils.rs:362:9
// note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
// test utils::tests::test_decompose_u64_digits_to_limbs ... FAILED
}
```

Figure 1.3: Test cases for the `decompose_biguint` and `decompose_u64_digits_to_limbs` functions

Exploit Scenario

An attacker submits values that do not satisfy the constraints of the system. However, since the values are incorrectly decomposed, the system accepts the results as satisfying the constraints.

Recommendations

Short term, for the `decompose_u64_digits_to_limbs` function, use a larger integer type such as `u128` to compute the mask to prevent an overflow from occurring when `bit_len` equals `64`. For the `decompose_biguint` function, have the function interpret `u64_digit` and `1` as `u128` before performing both the left-shift and right-shift operations. Add comprehensive unit tests for both functions.

Long term, investigate all other bit-shift operations that use variable length shifts with respect to the integer size that they operate on.

2. Risk of unconstrained inner product in release builds

Severity: **Medium**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-AXIOM-2
Target: `halo2-base/src/gates/flex_gate.rs`	

Description

The inner product API implementation in the `GateChip::inner_product_simple` gate uses the `size_hint` method to compute the number of assigned regions. This number is then used to compute the `gate_offsets` argument to the `Context::assign_region` function, which determines the offsets at which the gate should be enabled.

```
let gate_offsets = if ctx.witness_gen_only() {
    vec![]
} else {
    let (lo, hi) = cells.size_hint();
    debug_assert_eq!(Some(lo), hi);
    let len = lo / 3;
    (0..len).map(|i| 3 * i as isize).collect()
};
ctx.assign_region(cells, gate_offsets);
```

Figure 2.1: The implementation of the inner product API uses `size_hint` to determine the offsets at which the gate should be enabled.

The `size_hint` method is used to indicate the number of elements returned by the iterator. It returns lower and upper bounds on the number of elements returned. However, a valid implementation may return a lower bound that is strictly less than the actual size. Indeed, the default implementation of `size_hint` simply returns `(0, None)`, which is correct for any implementation.

The Rust documentation for `Iterator::size_hint` contains the following implementation note:

> *`size_hint` is primarily intended to be used for optimizations such as reserving space for the elements of the iterator, but must not be trusted to e.g., omit bounds checks in unsafe code. An incorrect implementation of `size_hint` should not lead to memory safety violations.*

If the `size_hint` implementation for either `a` or `b` returns a lower bound that is strictly smaller than the actual size of the iterator, the selector for the final region of the circuit will be disabled, and the cells in the final region will be unconstrained. In particular, this means that the value of the inner product will be unconstrained. The code checks for an unconstrained inner product by asserting that `Some(lo)` is equal to `hi`, but this `assert` statement is present in only debug builds, which means that the circuit could be underconstrained in release builds.

The same issue is present in the implementations of the following gates as well:

- `GateInstructions::sum`,
- `GateInstructions::partial_sums`
- `GateInstructions::select_by_indicator`
- `GateInstructions::select_from_idx`

The `GateChip::inner_product_left_last` gate also uses `size_hint` and will return the wrong cell if the output from `size_hint` is not equal to the size of the iterator `a.into()`.

Exploit Scenario
A developer builds a circuit that passes two custom iterators to the `GateChip` inner product API. Since the developer is not aware of how the API is implemented, he has not reimplemented the `size_hint` API on the custom iterators. When the circuit is constructed, `size_hint` falls back to the default implementation and `cells.size_hint()` returns `(0, None)`. Since the circuit is not properly tested, this behavior is not detected. It follows that the output of the inner product is unconstrained, which could allow malicious users to forge proofs.

Recommendations
Short term, replace the `debug_assert_eq` statement with a corresponding `assert_eq` statement after each use of `size_hint`. This will not be removed in release builds.

Long term, consider updating the `GateChip` API to take arguments implementing the `ExactSizeIterator` or `TrustedLen` instead.

3. idx_to_indicator circuit is underconstrained	
Severity: **High**	Difficulty: **Medium**
Type: Cryptography	Finding ID: TOB-AXIOM-3
Target: `halo2-base/src/gates/flex_gate.rs`	

Description

The `GateInstructions::idx_to_indicator` circuit is designed to constrain its output to a vector with a single bit set in the position indicated by the input `idx`. However, the circuit is missing an important constraint, which allows a malicious prover to set the output vector to a vector of all zeroes.

The `idx_to_indicator` circuit constraints, shown in figure 3.1, ensure that the output vector contains zero in every position except for `idx`. The circuit further constrains each element of the output vector to either zero or one. However, the circuit does not include a constraint that the output vector should be nonzero at the position specified by `idx`. Thus, for any input, an output vector of all zeroes is a satisfying assignment to the constraint system.

Similarly, when `idx` is not in the range `[0, len)`, the `idx_to_indicator` circuit returns an all-zero vector. When this vector is used in a dot-product as a selector, as in the `select_by_idx` function, it is ambiguous whether a zero result indicates a zero value in the target vector or an out-of-bounds index.

```
// returns vec with vec.len() == len such that:
//     vec[i] == 1{i == idx}
fn idx_to_indicator(
    &self,
    ctx: &mut Context<F>,
    idx: impl Into<QuantumCell<F>>,
    len: usize,
) -> Vec<AssignedValue<F>> {
    let mut idx = idx.into();
    let mut ind = Vec::with_capacity(len);
    let idx_val = idx.value().get_lower_32() as usize;
    for i in 0..len {
        // check ind[i] * (i - idx) == 0
        let ind_val = F::from(idx_val == i);
        let val = if idx_val == i { *idx.value() } else { F::zero() };
        ctx.assign_region_smart(
            vec![
                Constant(F::zero()),
```

```
                Witness(ind_val),
                idx,
                Witness(val),
                Constant(-F::from(i as u64)),
                Witness(ind_val),
                Constant(F::zero()),
            ],
            vec![0, 3],
            vec![(1, 5)],
            vec![],
        );
        // need to use assigned idx after i > 0 so equality constraint holds
        if i == 0 {
            idx = Existing(ctx.get(-5));
        }
        let ind_cell = ctx.get(-2);
        self.assert_bit(ctx, ind_cell);
        ind.push(ind_cell);
    }
    ind
}
```

Figure 3.1: `halo2-base/src/gates/flex_gate.rs#486–525`

Exploit Scenario

A developer uses the `select_by_idx` function to read elements at dynamic locations from an array, such as the value field of an RLP-serialized transaction. A malicious prover constructs an indicator vector consisting of all zeros, causing the `select_by_idx` circuit to prove that the transaction sent a zero value rather than the correct amount.

Recommendations

Short term, add an additional constraint for each indicator position specifying that the indicator value must be nonzero when the position equals the desired index.

Long term, in order to reduce ambiguity around zero outputs from `select_from_idx`, consider modifying `idx_to_indicator` and `select_from_idx` such that the circuit is unsatisfiable when the input index is beyond the provided length, providing a second output indicating whether the index was in-bounds, or allowing the caller to pass a desired default value.

4. ecdsa_verify_no_pubkey_check can fail on signatures from crafted public keys

Severity: **Medium**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-AXIOM-4
Target: `halo2-ecc/src/ecc/ecdsa.rs`	

Description

ECDSA signature verification requires that `u1*G + u2*PK != 0`, where `u1` and `u2` are intermediate values generated from an ECDSA signature. To perform this check, the ECDSA signature validation circuit compares the x-coordinates of `u1*G` and `u2*PK`:

```
// check u1 * G and u2 * pubkey are not negatives and not equal
//     TODO: Technically they could be equal for a valid signature, but this happens
with vanishing probability
//           for an ECDSA signature constructed in a standard way
// coordinates of u1_mul and u2_mul are in proper bigint form, and lie in but are
not constrained to [0, n)
// we therefore need hard inequality here
let u1_u2_x_eq = base_chip.is_equal(ctx, &u1_mul.x, &u2_mul.x);
let u1_u2_not_neg = base_chip.range.gate().not(ctx, u1_u2_x_eq);
```

Figure 4.1: The x-coordinate check

Although this check will prevent invalid `u1` and `u2` values, it will also prevent signatures in which `u1*G` and `u2*PK` are equal, which should be considered valid. As shown in figure 4.1, the comments for the check indicate that certain valid signatures may have equal x-coordinates but that this happens "with vanishing probability." However, if an attacker knows a message they will sign before they generate their secret key, they can select a secret key that allows them to create a signature for that chosen message where `u1*G == u2*PK`. In particular, given a chosen message hash `m = H(msg)` and an arbitrary nonce `k`, the secret key `sk = m*((kG).x)^-1` will generate a valid signature that will not satisfy this circuit.

Exploit Scenario

Suppose this circuit is used in an application such as a cross-chain bridge from chain A to chain B. A malicious user chooses a message `msg` that they will send and randomly samples the blinding factor `k` that they will use. They calculate a secret key `sk = H(msg)*((kG).x)^-1`, as described in the finding description. At a later point, that user submits `msg` to chain A with a signature based on the nonce `k`, which is accepted. However, that message and signature do not satisfy the circuit and cannot be bridged to chain B, leading to a denial of service.

Recommendations

Short term, modify the circuit so that it accepts `u1*G == u2*PK` but forbids `u1*G == -(u2*PK)`. For example, adding the check `(u1*G).x != (u2*PK).x OR (u1*G).y == (u2*PK).y` would resolve this issue.

Long term, ensure that optimizations that ignore low-probability events are still correct in adversarial situations.

5. log2_ceil function miscomputes its result when x input is zero

Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-AXIOM-5
Target: `halo2-lib/halo2-base/src/utils.rs`	

Description

When it receives a zero-value input, the `log2_ceil` function returns `u32::MAX` in release mode and panics due to an underflow in debug mode.

Figure 5.1 shows the function's implementation and its failure to handle an x input equal to zero. In that case, the expression `x - 1` will underflow in debug mode and will result in `u32::MAX` in release mode.

```
pub fn log2_ceil(x: u64) -> usize {
    (u64::BITS - x.leading_zeros() - (x & (x - 1) == 0) as u32) as usize
}
```

Figure 5.1: The `log2_ceil` implementation at `halo2-base/src/utils.rs#L101-L103`

The function is extensively used in `halo2-ecc` to compute the `max_limb_bits` field for the `OverflowInteger` type, which could cause miscalculations if certain arguments equal zero. For example, figure 5.2 shows the `scalar_mul_and_add_no_carry::assign` function, which will miscompute the maximum number of bits per limb when calculating `a * c + b`. If the argument `c` is zero, the `max_limb_bits` field of the result will equal `u32::MAX + a.max_limb_bits + 1` instead of `b.max_limb_bits + 1`.

```
/// compute a * c + b = b + a * c
// this is uniquely suited for our simple gate
pub fn assign<F: ScalarField>(
    gate: &impl GateInstructions<F>,
    ctx: &mut Context<F>,
    a: &OverflowInteger<F>,
    b: &OverflowInteger<F>,
    c_f: F,
    c_log2_ceil: usize,
) -> OverflowInteger<F> {
    debug_assert_eq!(a.limbs.len(), b.limbs.len());

    let out_limbs = a
        .limbs
        .iter()
        .zip(b.limbs.iter())
```

```
        .map(|(&a_limb, &b_limb)| gate.mul_add(ctx, a_limb, Constant(c_f), b_limb))
        .collect();

    OverflowInteger::construct(out_limbs, max(a.max_limb_bits + c_log2_ceil,
b.max_limb_bits) + 1)
}

/// compute a * c + b = b + a * c
pub fn crt<F: ScalarField>(
    gate: &impl GateInstructions<F>,
    ctx: &mut Context<F>,
    a: &CRTInteger<F>,
    b: &CRTInteger<F>,
    c: i64,
) -> CRTInteger<F> {
    debug_assert_eq!(a.truncation.limbs.len(), b.truncation.limbs.len());

    let (c_f, c_abs) = if c >= 0 {
        let c_abs = u64::try_from(c).unwrap();
        (F::from(c_abs), c_abs)
    } else {
        let c_abs = u64::try_from(-c).unwrap();
        (-F::from(c_abs), c_abs)
    };

    let out_trunc = assign::<F>(gate, ctx, &a.truncation, &b.truncation, c_f,
log2_ceil(c_abs));
    let out_native = gate.mul_add(ctx, a.native, Constant(c_f), b.native);
    let out_val = &a.value * c + &b.value;
    CRTInteger::construct(out_trunc, out_native, out_val)
}
```

Figure 5.2: The `scalar_mul_and_add_no_carry::assign` function, which will miscompute the maximum number of bits per limb if c is zero

Figure 5.3 shows another example in which the incorrect value could propagate into a range check.

```
pub fn truncate<F: BigPrimeField>(
    range: &impl RangeInstructions<F>,
    ctx: &mut Context<F>,
    a: OverflowInteger<F>,
    limb_bits: usize,
    limb_base: F,
    limb_base_big: &BigInt,
) {
    let k = a.limbs.len();
    let max_limb_bits = a.max_limb_bits;

    let mut carries = Vec::with_capacity(k);

    for a_limb in a.limbs.iter() {
```

```
        let a_val_big = fe_to_bigint(a_limb.value());
        let carry = if let Some(carry_val) = carries.last() {
            (a_val_big + carry_val) / limb_base_big
        } else {
            // warning: using >> on negative integer produces undesired effect
            a_val_big / limb_base_big
        };
        carries.push(carry);
    }

    // round `max_limb_bits - limb_bits + EPSILON + 1` up to the next multiple of
range.lookup_bits
    const EPSILON: usize = 1;
    let range_bits = max_limb_bits - limb_bits + EPSILON;
    let range_bits =
        ((range_bits + range.lookup_bits()) / range.lookup_bits()) *
range.lookup_bits() - 1;
    // `window = w + 1` valid as long as `range_bits + n * (w+1) <
native_modulus::<F>().bits() - 1`
    // let window = (F::NUM_BITS as usize - 2 - range_bits) / limb_bits;
    // assert!(window > 0);
    // In practice, we are currently always using window = 1 so the above is
commented out

    let shift_val = range.gate().pow_of_two()[range_bits];
    // let num_windows = (k - 1) / window + 1; // = ((k - 1) - (window - 1) + window
- 1) / window + 1;

    let mut previous = None;
    for (a_limb, carry) in a.limbs.into_iter().zip(carries.into_iter()) {
        let neg_carry_val = bigint_to_fe(&-carry);
        ctx.assign_region(
            [
                Existing(a_limb),
                Witness(neg_carry_val),
                Constant(limb_base),
                previous.map(Existing).unwrap_or_else(|| Constant(F::zero())),
            ],
            [0],
        );
        let neg_carry = ctx.get(-3);

        // i in 0..num_windows {
        // let idx = std::cmp::min(window * i + window - 1, k - 1);
        // let carry_cell = &neg_carry_assignments[idx];
        let shifted_carry = range.gate().add(ctx, neg_carry, Constant(shift_val));
        range.range_check(ctx, shifted_carry, range_bits + 1);

        previous = Some(neg_carry);
    }
}
```

Figure 5.3: `halo2-ecc/src/bigint/check_carry_to_zero.rs#L27-L86`

Exploit Scenario

A new operation is implemented with constraints related to the maximum limb bit-size. The miscalculation of the `log2_ceil` value leads to an unnecessarily large number of constraints or truncates the number of added constraints, leading to an underconstrained circuit.

Recommendations

Short term, have the `log2_ceil` function either panic or return zero when called with a zero-value input for `x`; either of these options would suit the use cases of the function. Add documentation for this case and comprehensive tests for the `bigint` library operations.

6. GateChip::num_to_bits depends on implementation-specific details of the underlying field	
Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-AXIOM-6
Target: `halo2-base/src/gates/flex_gate.rs`	

Description

The implementation of the `num_to_bits` function calls the `to_repr` function on the value of the input a when constructing the little-endian binary representation of the input a.

```
// returns little-endian bit vectors
fn num_to_bits(
    &self,
    ctx: &mut Context<F>,
    a: AssignedValue<F>,
    range_bits: usize,
) -> Vec<AssignedValue<F>> {
    let a_bytes = a.value().to_repr();
    let bits = a_bytes
        .as_ref()
        .iter()
        .flat_map(|byte| (0..8).map(|i| (*byte as u64 >> i) & 1))
        .take(range_bits)
        .map(|x| F::from(x));

    // ... <redacted>
}
```

Figure 6.1: The implementation of `num_to_bits` expects `to_repr` to return a little-endian representation of the value of a.

However, according to the documentation of the `PrimeField` trait, the endianness returned by `PrimeField::to_repr` is implementation-dependent and may be different depending on the underlying field.

```
/// Converts an element of the prime field into the standard byte representation for
/// this field.
///
/// The endianness of the byte representation is implementation-specific. Generic
/// encodings of field elements should be treated as opaque.
fn to_repr(&self) -> Self::Repr;
```

Figure 6.2: The value returned by `to_repr` is implementation-dependent and should be treated as opaque by the user.

The same issue is present in the implementations of the following 10 functions:

- `unpack` (in `hashes/zkevm-keccak/src/util.rs`)
- `U256::to_scalar` (in `hashes/zkevm-keccak/src/util/eth_types.rs`)
- `Address::to_scalar` (in `hashes/zkevm-keccak/src/util/eth_types.rs`)
- `fe_to_biguint` (in `halo2-base/src/utils.rs`)
- `biguint_to_fe` and `bigint_to_fe` (in `halo2-base/src/utils.rs`)
- `fe_from_big`, `fe_to_big`, `fe_from_limbs`, and `fe_to_limbs` (in `snark-verifier/src/util/arithmetic.rs`)
- `EthBlockHeaderChainInstance::from_instance` (in `axiom-eth/src/block_header/mod.rs`)

Exploit Scenario

The `GateChip` implementation is reused with a scalar field F that uses a different internal representation of the elements of F. This means that the generated witness will not satisfy the circuit.

Recommendations

Short term, replace each use of `PrimeField::to_repr` with `ScalarField::to_u64_limbs`, which guarantees that the returned value will be little-endian.

Long term, review the use of third-party APIs to ensure that the codebase does not depend on the internal representation of data.

7. RangeChip::get_last_bit returns the wrong value	
Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-AXIOM-7
Target: `halo2-base/src/gates/range.rs`	

Description

The `RangeChip::get_last_bit` function should return the least significant bit of the input a, properly constrained. During witness generation, the least significant bit of a, `bit_v`, is computed as follows:

```
let bit_v = {
    let a = a_v.get_lower_32();
    F::from(a ^ 1 != 0)
};
```

Figure 7.1: The value of `bit_v` is not equal to the least significant bit of a.

Based on the implementation, this means that the value of `bit_v` is given by the following:

$$\texttt{bit\_v} = \begin{cases} \texttt{F::from(1)}, \text{ if the least significant 32 bits of a are greater than 1} \\ \texttt{F::from(0)}, \text{ if the least significant 32 bits of a are equal to 1} \\ \texttt{F::from(1)}, \text{ if the least significant 32 bits of a are equal to 0} \end{cases}$$

This means that the value of `bit_v` will not equal the least significant bit of a, and the generated witness will not satisfy the constraints defined by the circuit.

Moreover, the return value of the `get_last_bit` function is given by the `bit` variable, which is defined as follows:

```
let two = self.gate().get_field_element(2u64);
let h_v = (*a_v - bit_v) * two.invert().unwrap();
ctx.assign_region(
    vec![Witness(bit_v), Witness(h_v), Constant(two), Existing(a)],
    Vec![0]
);

let half = ctx.get(-3);
self.range_check(ctx, half, limb_bits - 1);
let bit = ctx.get(-4);
self.gate().assert_bit(ctx, bit);
```

```
bit
```

Figure 7.2: The call to `ctx.get` must occur before the range check for the cell `half`, but it occurs after the check.

However, the call to `ctx.get`, shown in figure 7.2, obtains a cell from the range check for the `half` variable rather than the assigned cell corresponding to `Witness(bit_v)`, as intended; this is because the call to `ctx.get` occurs after the range check for the cell `half`, but it should occur before the check. This cell is then constrained to be binary before it is returned. This means that the function will return the wrong value, and the least significant bit of a will be underconstrained.

The following is a failing unit test for `RangeChip::get_last_bit`.

```
#[test]
fn test_last_bit_value() {
    // Create a builder and obtain a context.
    let mut builder = GateThreadBuilder::new(false);
    let ctx = builder.main(0);

    // Create a new RangeChip.
    let chip: RangeChip<Fr> = RangeChip::new(RangeStrategy::Vertical, 8);

    // Get the least significant bit of 3 (which should be 1).
    let a = ctx.assign_witnesses([Fr::from(3)])[0];
    let bit = chip.get_last_bit(ctx, a, 8);

    assert_eq!(*bit.value(), Fr::from(1));
}
// running 1 test
// thread 'gates::range::tests::test_last_bit_value' panicked at 'assertion failed:
// `(left == right)`
//   left: `0x0000000000000000000000000000000000000000000000000000000000000000`,
//  right: `0x0000000000000000000000000000000000000000000000000000000000000001`'
```

Figure 7.3: Running the unit test shows that `get_last_bit` returns `0` on input 3.

Recommendations

Short term, replace the computation of `bit_v` with the following:

```
let bit_v = {
    let a = a_v.get_lower_32() as u64;
    F::from(a & 1)
};
```

Figure 7.4: The correct definition of `bit_v`

Ensure that the definition of `bit` occurs before the range check for `half` to ensure that the index given to `ctx.get` is still valid.

Long term, ensure that each of the functions defined by `GateInstructions` and `RangeInstructions` has a corresponding unit test, checking that the output from each function is correct.

8. Validations missing in release builds	
Severity: **Medium**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-AXIOM-8
Target: Several files	

Description

Axiom's Halo2 libraries extensively use debug assertions for data and invariant validations. Because they are defined as debug assertions, these validations will not be present in the release builds, which could lead to incorrect results or runtime errors.

For example, the `BaseConstraintBuilder::validate_degree` function validates the constraint degree only in debug mode:

```
pub(crate) fn validate_degree(&self, degree: usize, name: &'static str) {
    if self.max_degree > 0 {
        debug_assert!(
            degree <= self.max_degree,
            "Expression {} degree too high: {} > {}",
            name,
            degree,
            self.max_degree,
        );
    }
}
```

Figure 8.1: `hashes/zkevm-keccak/src/util/constraint_builder.rs#L54-L64`

Thus, in release mode, the `filter` present in the `BaseConstraintBuilder::gate` function will effectively be a no-op:

```
   pub(crate) fn gate(&self, selector: Expression<F>) -> Vec<(&'static str, Expression<F>)> {
        self.constraints
            .clone()
            .into_iter()
            .map(|(name, constraint)| (name, selector.clone() * constraint))
            .filter(|(name, constraint)| {
                self.validate_degree(constraint.degree(), name);
                true
            })
            .collect()
    }
}
```

Figure 8.2: `hashes/zkevm-keccak/src/util/constraint_builder.rs#L66-L77`

Figure 8.3 shows another example of this issue in the `sub::assign` function. The implementation uses a debug assertion to check that the arguments have the same number of limbs, but this requirement is not documented. If this function is used with `OverflowIntegers` of different numbers of limbs, it will compute the incorrect result of the subtraction because it will ignore limbs from the longer-limbed `OverflowInteger`. This behavior contrasts with the behavior of the `mul_no_carry::truncate` function, which requires that both arguments have the same number of limbs.

```
/// Should only be called on integers a, b in proper representation with all limbs
having at most `limb_bits` number of bits
pub fn assign<F: ScalarField>(
    range: &impl RangeInstructions<F>,
    ctx: &mut Context<F>,
    a: &OverflowInteger<F>,
    b: &OverflowInteger<F>,
    limb_bits: usize,
    limb_base: F,
) -> (OverflowInteger<F>, AssignedValue<F>) {
    debug_assert!(a.max_limb_bits <= limb_bits);
    debug_assert!(b.max_limb_bits <= limb_bits);
    debug_assert_eq!(a.limbs.len(), b.limbs.len());
    let k = a.limbs.len();
    let mut out_limbs = Vec::with_capacity(k);

    let mut borrow: Option<AssignedValue<F>> = None;
    for (&a_limb, &b_limb) in a.limbs.iter().zip(b.limbs.iter()) {
```

Figure 8.3: `halo2-ecc/src/bigint/sub.rs#L9-L25`

In the implementation of `bigint` operations, different functions verify that both arguments have the same number of limbs in three different ways:

- There is a non-debug assertion (e.g., on `mul_no_carry::truncate`). This will always validate the intended behavior.
- There is a debug assertion and no documentation. An API user would not know about this intended check and might call the function in ways that are unintended, causing incorrect results or runtime errors when it is used in release mode.
- There is a debug assertion and documentation. The API user has the responsibility to validate the function preconditions.

If API misuse could cause incorrect values that would propagate without a runtime error, as in the `sub::assign` function, it is recommended to enforce the validation with a non-debug assertion.

Recommendations

Short term, add validation to all debug assertions and document the invariants for API users, or add non-debug assertions that will perform the validation on release builds in cases that could compute incorrect values or pose security issues.

9. Keccak implementation cannot hash arbitrarily large inputs	
Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-AXIOM-9
Target: `hashes/zkevm-keccak/src/keccak_packed_multi.rs`	

Description

The Keccak implementation will result in a runtime error if it tries to hash more than `isize::MAX/8` bytes. This issue affects the PSE implementation that targets WASM builds and instances in which pointers are 32 bits (i.e., `isize::MAX` = 2^{31}`-1`). This means that the Keccak WASM implementation is unable to hash more than approximately 270 MB without having a runtime error.

The `keccak_phase0` function, shown in figure 9.1, converts the `bytes` array argument into a `bits` vector that is eight times the size of the `bytes` array. Then, padding is added according to the Keccak specification. However, `Vec::push` will panic if the capacity exceeds `isize::MAX`. This means that in a 32-bit system, the bytes argument cannot have more than `(`2^{31}`-3)/8` elements, which is approximately 270 MB.

```
/// Witness generation in `FirstPhase` for a keccak hash digest without
/// computing RLCs, which are deferred to `SecondPhase`.
pub fn keccak_phase0<F: Field>(
    rows: &mut Vec<KeccakRow<F>>,
    squeeze_digests: &mut Vec<[F; NUM_WORDS_TO_SQUEEZE]>,
    bytes: &[u8],
) {
    let mut bits = into_bits(bytes);
    let mut s = [[F::zero(); 5]; 5];
    let absorb_positions = get_absorb_positions();
    let num_bytes_in_last_block = bytes.len() % RATE;
    let num_rows_per_round = get_num_rows_per_round();
    let two = F::from(2u64);

    // Padding
    bits.push(1);
    while (bits.len() + 1) % RATE_IN_BITS != 0 {
        bits.push(0);
    }
    bits.push(1);
```

Figure 9.1: `hashes/zkevm-keccak/src/keccak_packed_multi.rs#L1636-L1655`

Recommendations

Short term, add test vectors for Keccak, including tests that include large messages; document the potential limitations of a WASM build.

10. Field division of zero by zero is unconstrained

Severity: **Medium**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-AXIOM-10
Target: `halo2-ecc/src/fields/mod.rs`	

Description

The `FieldChip::divide` and `FieldChip::neg_divide` functions do not check that b is nonzero. If both field elements a and b equal zero, a malicious prover could select any value for the witness value q.

In the `fields` module of `halo2-ecc`, the `FieldChip` trait defines an interface for finite field arithmetic and includes some default implementations for common operations.

Given the field elements a and b, `FieldChip::divide` and `FieldChip::neg_divide` calculate ab^{-1} and $-\left(ab^{-1}\right)$, respectively, by returning a witness value q, which is constrained by the equations $qb - a \equiv 0$ and $qb + a \equiv 0$, respectively:

```
fn divide(
    &self,
    ctx: &mut Context<F>,
    a: &Self::FieldPoint,
    b: &Self::FieldPoint,
) -> Self::FieldPoint {
    let a_val = self.get_assigned_value(a);
    let b_val = self.get_assigned_value(b);
    let b_inv = b_val.invert().unwrap();
    let quot_val = a_val * b_inv;

    let quot = self.load_private(ctx, Self::fe_to_witness(&quot_val));

    // constrain quot * b - a = 0 mod p
    let quot_b = self.mul_no_carry(ctx, &quot, b);
    let quot_constraint = self.sub_no_carry(ctx, &quot_b, a);
    self.check_carry_mod_to_zero(ctx, &quot_constraint);

    quot
}
```

Figure 10.1: `halo2-ecc/src/fields/mod.rs#165–184`

```
// constrain and output -a / b
// this is usually cheaper constraint-wise than computing -a and then (-a) / b
separately
```

```rust
fn neg_divide(
    &self,
    ctx: &mut Context<F>,
    a: &Self::FieldPoint,
    b: &Self::FieldPoint,
) -> Self::FieldPoint {
    let a_val = self.get_assigned_value(a);
    let b_val = self.get_assigned_value(b);
    let b_inv = b_val.invert().unwrap();
    let quot_val = -a_val * b_inv;

    let quot = self.load_private(ctx, Self::fe_to_witness(&quot_val));
    self.range_check(ctx, &quot, Self::PRIME_FIELD_NUM_BITS as usize);

    // constrain quot * b + a = 0 mod p
    let quot_b = self.mul_no_carry(ctx, &quot, b);
    let quot_constraint = self.add_no_carry(ctx, &quot_b, a);
    self.check_carry_mod_to_zero(ctx, &quot_constraint);

    quot
}
```

Figure 10.2: `halo2-ecc/src/fields/mod.rs#186–208`

If $b = 0$ and $a \neq 0$, these equations are unsatisfiable. However, if $a = 0$ and $b = 0$, a malicious prover could select any value for q.

Exploit Scenario

A developer uses `divide` to calculate $q = ab^{-1}$ within a circuit. They test the behavior of `divide` on a few inputs and conclude that $b = 0$ will be rejected. They optimize their circuit by skipping a check for $b \neq 0$. A malicious prover can then set $(a, b) = (0, 0)$ and arbitrarily choose the value of q, forging a proof.

Recommendations

Short term, constrain b to be nonzero in both `FieldChip::divide` and `FieldChip::neg_divide`.

Long term, require inline documentation to specify edge-case behaviors that must be validated. Ensure that critical functions have this documentation and that the implementation follows these recommendations.

11. Incorrect point-at-infinity handling in elliptic curve operations

Severity: **Medium**	Difficulty: **Low**
Type: Cryptography	Finding ID: TOB-AXIOM-11
Target: `halo2-ecc/src/ecc/mod.rs`	

Description

Some elliptic curve operations do not correctly handle the point at infinity and return incorrect results.

The `ecc` module of `halo2-ecc` implements elliptic curve arithmetic operations within a `halo2` circuit. Within `ecc`, elliptic curve points are represented in affine coordinates as an `(x,y)` pair:

```
// EcPoint and EccChip take in a generic `FieldChip` to implement generic elliptic
curve operations on arbitrary field extensions (provided chip exists) for short
Weierstrass curves (currently further assuming a4 = 0 for optimization purposes)
#[derive(Debug)]
pub struct EcPoint<F: PrimeField, FieldPoint> {
    pub x: FieldPoint,
    pub y: FieldPoint,
    _marker: PhantomData<F>,
}
```

Figure 11.1: `halo2-ecc/src/ecc/mod.rs#23–29`

Although the `EcPoint` struct does not have a documented representation for the point at infinity, several components treat `(0,0)` as the point at infinity:

```
pub fn is_on_curve_or_infinity<C>(
    &self,
    ctx: &mut Context<F>,
    P: &EcPoint<F, FC::FieldPoint>,
) -> AssignedValue<F>
where
    C: CurveAffine<Base = FC::FieldType>,
    C::Base: ff::PrimeField,
{
...
    let is_on_curve = self.field_chip.is_zero(ctx, &diff);

    let x_is_zero = self.field_chip.is_zero(ctx, &P.x);
    let y_is_zero = self.field_chip.is_zero(ctx, &P.y);
```

```
    self.field_chip.range().gate().or_and(ctx, is_on_curve, x_is_zero, y_is_zero)
}
```

Figure 11.2: `halo2-ecc/src/ecc/mod.rs#661–685`

The public functions `ec_add_unequal`, `ec_sub_unequal`, and `ec_double_and_add_unequal` do not explicitly handle this input, so they can return incorrect results if given the point at infinity. For example, if `ec_add_unequal` is called with $P = (0, 0)$ and $Q = \left(x_Q, y_Q\right)$, then `ec_add_unequal` will compute $\lambda \equiv \frac{y_Q}{x_Q}$, $x \equiv \lambda^2 - x_Q$, and $y \equiv \ - \lambda x$, instead of $x \equiv x_Q$ and $y \equiv y_Q$:

```
let dx = chip.sub_no_carry(ctx, &Q.x, &P.x);
let dy = chip.sub_no_carry(ctx, &Q.y, &P.y);
let lambda = chip.divide(ctx, &dy, &dx);

//  x_3 = lambda^2 - x_1 - x_2 (mod p)
let lambda_sq = chip.mul_no_carry(ctx, &lambda, &lambda);
let lambda_sq_minus_px = chip.sub_no_carry(ctx, &lambda_sq, &P.x);
let x_3_no_carry = chip.sub_no_carry(ctx, &lambda_sq_minus_px, &Q.x);
let x_3 = chip.carry_mod(ctx, &x_3_no_carry);

//  y_3 = lambda (x_1 - x_3) - y_1 mod p
let dx_13 = chip.sub_no_carry(ctx, &P.x, &x_3);
let lambda_dx_13 = chip.mul_no_carry(ctx, &lambda, &dx_13);
let y_3_no_carry = chip.sub_no_carry(ctx, &lambda_dx_13, &P.y);
let y_3 = chip.carry_mod(ctx, &y_3_no_carry);

EcPoint::construct(x_3, y_3)
```

Figure 11.3: The calculation of `ec_add_unequal` in `halo2-ecc/src/ecc/mod.rs#75–91`

The following are the affected functions:

- `ec_add_unequal` in `halo2-ecc/src/ecc/mod.rs#51-68`
- `ec_sub_unequal` in `halo2-ecc/src/ecc/mod.rs#94-110`
- `ec_double_and_add_unequal` in `halo2-ecc/src/ecc/mod.rs#180-195`

Some other elliptic curve functions have comments ruling out the point at infinity. Any uses of these functions should be carefully reviewed to prevent unsafe usage:

- The comments for `ec_double` state, "`assume y != 0 (otherwise 2P = O)`" (`halo2-ecc/src/ecc/mod.rs#142-158`).
- The comments for `scalar_multiply` state that it assumes "P has `order given by the scalar field modulus`" (`halo2-ecc/src/ecc/mod.rs#289-306`).

- The comments for `multi_scalar_multiply` state that it makes the same assumptions as `scalar_multiply` (`halo2-ecc/src/ecc/mod.rs#425-441`).

Exploit Scenario

A developer implements a new component using the `ecc` module's elliptic curve operations. The developer is unfamiliar with this issue and implements input validation by calling `EccChip::is_on_curve_or_infinity`. A malicious prover can then cause the operations to return incorrect results by providing the point at infinity as input, causing the component to misbehave and potentially forging a proof.

Recommendations

Short term, have all elliptic curve operations explicitly handle point-at-infinity inputs, or document contexts in which those inputs are not allowed.

Long term, require inline documentation to specify edge-case behaviors that must be validated. Ensure that critical functions have this documentation and that the implementation follows these recommendations.

12. FpChip::load_private allows non-reduced field elements	
Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-AXIOM-12
Target: `halo2-ecc/src/fields/fp.rs`	

Description

The `FpChip::load_private` function does not require its return value to be below the field modulus p.

In the `fields` module of `halo2-ecc`, finite field witness values are added to the circuit via the `FpChip::load_private` function. `FpChip::load_private` converts a `BigInt` value into a `CRTInteger` value and calls the `range_check` function to ensure that the `truncation` field of the witness value is in the proper reduced form:

```
fn load_private(&self, ctx: &mut Context<F>, a: BigInt) -> CRTInteger<F> {
    let a_vec = decompose_bigint::<F>(&a, self.num_limbs, self.limb_bits);
    let limbs = ctx.assign_witnesses(a_vec);

    let a_native = OverflowInteger::<F>::evaluate(
        self.range.gate(),
        ctx,
        limbs.iter().copied(),
        self.limb_bases.iter().copied(),
    );

    let a_loaded =
        CRTInteger::construct(OverflowInteger::construct(limbs, self.limb_bits),
a_native, a);

    // TODO: this range check prevents loading witnesses that are not in "proper"
representation form, is that ok?
    self.range_check(ctx, &a_loaded, Self::PRIME_FIELD_NUM_BITS as usize);
    a_loaded
}
```

Figure 12.1: `halo2-ecc/src/fields/fp.rs#144–161`

The `range_check` call highlighted in figure 12.1 guarantees that the value of `a_loaded.truncation` will have reduced limbs and will be in the range $[0, 2^{PRIME\_FIELD\_NUM\_BITS})$, but it does not ensure that it is below the field modulus p. A malicious prover could load a witness that is in the range $[p, 2^{PRIME\_FIELD\_NUM\_BITS})$, potentially causing other circuit components to misbehave.

Exploit Scenario

A developer loads a scalar value s into the circuit using `FpChip::load_private`. Assuming that s is already in reduced form, they do not perform an additional bounds check, and they pass s directly into a component that assumes its inputs are reduced. A malicious prover can then choose a value of s that is between p and $2^{PRIME\_FIELD\_NUM\_BITS}$, violating preconditions and potentially forging a proof.

Recommendations

Short term, add an additional check to ensure that loaded values are below p.

Long term, document assumptions and guarantees about when field elements are in reduced form. Consider using the Rust newtype pattern to track this information in the type system and catch errors at compile time.

13. scalar_multiply can return underconstrained results

Severity: **High**	Difficulty: **Medium**
Type: Cryptography	Finding ID: TOB-AXIOM-13
Target: `halo2-ecc/src/ecc/{mod,fixed_base}.rs`	

Description

The results of the `ecc::scalar_multiply` and `fixed_base::scalar_multiply` functions will be underconstrained when the functions are given certain scalar values.

In `halo2-ecc`, the `scalar_multiply` function multiplies an elliptic curve point P by a scalar value s using a windowed algorithm with 2^k-sized chunks. First, it builds a cache of values of iP for $i = 1, 2, 3, \ldots, (2^k - 1)$:

```
// cached_points[idx] stores idx * P, with cached_points[0] = P
let cache_size = 1usize << window_bits;
let mut cached_points = Vec::with_capacity(cache_size);
cached_points.push(P.clone());
cached_points.push(P.clone());
for idx in 2..cache_size {
    if idx == 2 {
        let double = ec_double(chip, ctx, P /*, b*/);
        cached_points.push(double);
    } else {
        let new_point = ec_add_unequal(chip, ctx, &cached_points[idx - 1], P,
false);
        cached_points.push(new_point);
    }
}
```

Figure 13.1: The code that builds the point cache (`halo2-ecc/src/ecc/mod.rs#344–357`)

Then, for each value of i in the sequence $0, 1, \ldots, (num\_windows - 1)$, it calculates $R_{i+1} = 2^k R_i + c_i P$ for each k-bit chunk c_i of s. When $c_i \neq 0$, the circuit adds $2^k R_i$ to $c_i P$ by calling the `ec_add_unequal` function with its `is_strict` parameter set to `false`:

```
for idx in 1..num_windows {
    let mut mult_point = curr_point.clone();
    for _ in 0..window_bits {
        mult_point = ec_double(chip, ctx, &mult_point);
    }
    let add_point = ec_select_from_bits::<F, FC>(
        chip,
```

```
        ctx,
        &cached_points,
        &rounded_bits
            [rounded_bitlen - window_bits * (idx + 1)..rounded_bitlen - window_bits
* idx],
    );
    let mult_and_add = ec_add_unequal(chip, ctx, &mult_point, &add_point, false);
    let is_started_point =
        ec_select(chip, ctx, &mult_point, &mult_and_add, is_zero_window[idx]);

    curr_point =
        ec_select(chip, ctx, &is_started_point, &add_point, is_started[window_bits *
idx]);
}
```

Figure 13.2: The double-and-add loop (`halo2-ecc/src/ecc/mod.rs#367–385`)

When `is_strict` is `false`, `ec_add_unequal` skips a check requiring $P.x$ and $Q.x$ to be unequal. Figure 13.3 shows that if $P = \ - Q$, the intermediate value `lambda` will be calculated by dividing $2(P.x)$ by zero, which will be unsatisfiable. However, if $P = Q$, it will compute `lambda` by dividing zero by zero. Due to the issue described in finding TOB-AXIOM-10, `lambda` can be set arbitrarily by the prover, which could lead to an overall incorrect result:

```
if is_strict {
    // constrains that P.x != Q.x
    let x_is_equal = chip.is_equal_unenforced(ctx, &P.x, &Q.x);
    chip.range().gate().assert_is_const(ctx, &x_is_equal, &F::zero());
}

let dx = chip.sub_no_carry(ctx, &Q.x, &P.x);
let dy = chip.sub_no_carry(ctx, &Q.y, &P.y);
let lambda = chip.divide(ctx, &dy, &dx);
```

Figure 13.3: `ec_add_unequal` (`halo2-ecc/src/ecc/mod.rs#69–77`)

If P is an order-n curve point, a malicious prover can exploit the skipped check described in the previous paragraph by finding a and b integers such that $a \geq 1, 1 \leq b < 2^k$, and $a2^k \equiv b \bmod n$, and setting $scalar = a2^k + b$. If $\left(n \bmod 2^k\right) \neq 0$, then (a, b) can be calculated by setting $b = 2^k - \left(n \bmod 2^k\right) a = \frac{n+b}{2^k}$.

Requiring that $scalar \in [1, n)$ would be sufficient to prevent this issue, since $b < a2^k < n$.

Exploit Scenario
A developer uses `scalar_multiply` to implement a signature scheme without checking that the scalar is fully reduced modulo n, and a malicious prover is able to trigger this bug and then arbitrarily choose a value for `lambda` within `ec_add_unequal`. For example, note the lines of `ecdsa_verify_no_pubkey_check` highlighted in figure 13.4 below. Due to the

issue described in finding TOB-AXIOM-12, a malicious prover can cause the value of `u2` to be larger than p. If the red-highlighted `big_less_than` check were removed, that malicious prover would be able to exploit `scalar_multiply` and cause an incorrect value of `u2_mul` to be calculated.

```
// compute u1 = m s^{-1} mod n and u2 = r s^{-1} mod n
let u1 = scalar_chip.divide(ctx, msghash, s);
let u2 = scalar_chip.divide(ctx, r, s);

//let r_crt = scalar_chip.to_crt(ctx, r)?;

// compute u1 * G and u2 * pubkey
let u1_mul = fixed_base::scalar_multiply::<F, _, _>(
    base_chip,
    ctx,
    &GA::generator(),
    u1.truncation.limbs.clone(),
    base_chip.limb_bits,
    fixed_window_bits,
);
let u2_mul = scalar_multiply::<F, _>(
    base_chip,
    ctx,
    pubkey,
    u2.truncation.limbs.clone(),
    base_chip.limb_bits,
    var_window_bits,
);

…

// TODO: maybe the big_less_than is optional?
let u1_small = big_less_than::assign::<F>(
    base_chip.range(),
    ctx,
    &u1.truncation,
    &n.truncation,
    base_chip.limb_bits,
    base_chip.limb_bases[1],
);
let u2_small = big_less_than::assign::<F>(
    base_chip.range(),
    ctx,
    &u2.truncation,
    &n.truncation,
    base_chip.limb_bits,
    base_chip.limb_bases[1],
);
```

Figure 13.4: `u2_mul` may be calculated incorrectly if the `u2_small` bounds check is removed. (`halo2-ecc/src/ecc/ecdsa.rs#37–93`)

Recommendations

Short term, modify `scalar_multiply` to ensure that `ec_add_unequal` is not called with equal inputs when `is_strict == false`. For example, having the function set `is_strict` to `true` would resolve this issue.

Long term, consider replacing uses of `ec_add_unequal` with a more general elliptic curve addition function that explicitly handles edge cases like point doubling, inverses, and the point at infinity.

14. Witness may be underconstrained if two gates overlap with more than one cell

Severity: **Informational**	Difficulty: **N/A**
Type: Data Validation	Finding ID: TOB-AXIOM-14
Target: `halo2-base/src/gates/builder.rs`	

Description

The `GateThreadBuilder::assign_all` method assigns advice cells, enables selectors, and enforces copy constraints. When the end of an advice column is reached, the witness assignment process continues with the next advice column from the next basic gate. To ensure that values are preserved, the last cell of column `m` is repeated as the first cell of column `m` + 1, and a copy constraint is added to ensure that the two cells are equal. This is done to account for the case in which two basic gates overlap.

```
if (q && row_offset + 4 > max_rows) || row_offset >= max_rows - 1 {
    break_point.push(row_offset);
    row_offset = 0;
    gate_index += 1;

    // when there is a break point, because we may have two gates that overlap at
    // the current cell, we must copy the current cell to the next column for safety
    basic_gate = config.basic_gates[phase]
    .get(gate_index)
    .unwrap_or_else(|| panic!(
        "NOT ENOUGH ADVICE COLUMNS IN PHASE {phase}. Perhaps blinding factors
        were not taken into account. The max non-poisoned rows is {max_rows}"
    ));
    let column = basic_gate.value;

    #[cfg(feature = "halo2-axiom")]
    {
        let ncell =
            *region.assign_advice(column, row_offset, value).unwrap().cell();
        region.constrain_equal(&ncell, &cell);
    }
    #[cfg(not(feature = "halo2-axiom"))]
    {
        let ncell = region
            .assign_advice(|| "", column, row_offset, || value)
            .unwrap()
            .cell();
        region.constrain_equal(ncell, cell).unwrap();
    }
```

```
}
```

Figure 14.1: A copy constraint is introduced to ensure that the first cell of column `m + 1` (`cell`) is equal to the last cell of column `m` (`cell`).

However, if a developer were to add two basic gates that overlap with more than one cell, then only the first overlapping cell would be assigned to column `m` before switching to column `m + 1`. This means that the last basic gate in column `m` would not be complete, and the remaining overlapping cells would be underconstrained.

There are currently no such gates defined in the codebase, but there is nothing in the API stopping a developer from adding one.

Recommendations

Short term, have the `assign_all` method check that basic gates overlap with only one cell and panic if the overlap is more than one cell.

15. EccChip::load_private does not enforce that witness values are on-curve

Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-AXIOM-15
Target: `halo2-ecc/src/ecc/mod.rs`	

Description

The `EccChip::load_private` function may allow the prover to choose an off-curve point as a witness for elliptic curve operations.

The `EccChip` component of `halo2_ecc` provides the `load_private` and `assign_point` methods to load an elliptic curve point into a circuit. As shown in figure 15.1, these methods load field elements for the x- and y-coordinates without checking that the loaded coordinate pair is on the curve:

```
pub fn load_private(
    &self,
    ctx: &mut Context<F>,
    point: (FC::FieldType, FC::FieldType),
) -> EcPoint<F, FC::FieldPoint> {
    let (x, y) = (FC::fe_to_witness(&point.0), FC::fe_to_witness(&point.1));

    let x_assigned = self.field_chip.load_private(ctx, x);
    let y_assigned = self.field_chip.load_private(ctx, y);

    EcPoint::construct(x_assigned, y_assigned)
}

/// Does not constrain witness to lie on curve
pub fn assign_point<C>(&self, ctx: &mut Context<F>, g: C) -> EcPoint<F,
FC::FieldPoint>
where
    C: CurveAffineExt<Base = FC::FieldType>,
{
    let (x, y) = g.into_coordinates();
    self.load_private(ctx, (x, y))
}
```

Figure 15.1: `halo2-ecc/src/ecc/mod.rs#613–633`

These functions are then used within other witness-loading functions such as `load_private_g1` and `load_private_g2` in the `PairingChip` component:

```
pub fn load_private_g1(&self, ctx: &mut Context<F>, point: G1Affine) -> EcPoint<F,
```

```
FpPoint<F>> {
    let g1_chip = EccChip::new(self.fp_chip);
    g1_chip.load_private(ctx, (point.x, point.y))
}

pub fn load_private_g2(
    &self,
    ctx: &mut Context<F>,
    point: G2Affine,
) -> EcPoint<F, FieldExtPoint<FpPoint<F>>> {
    let fp2_chip = Fp2Chip::<F>::new(self.fp_chip);
    let g2_chip = EccChip::new(&fp2_chip);
    g2_chip.load_private(ctx, (point.x, point.y))
}
```

Figure 15.2: `halo2-ecc/src/bn254/pairing.rs#448–461`

The `EccChip::assign_point` function is used to implement several functions in `snark-verifier`. The `<BaseFieldEccChip<C> as EccInstructions<C>>::assign_point` function, shown in figure 15.3, checks that the point is on the curve or infinity:

```
fn assign_point(&self, ctx: &mut Self::Context, point: C) -> Self::AssignedEcPoint {
    let assigned = self.assign_point(ctx.main(0), point);
    let is_valid = self.is_on_curve_or_infinity::<C>(ctx.main(0), &assigned);
    self.field_chip().gate().assert_is_const(ctx.main(0), &is_valid,
&C::Scalar::one());
    assigned
}
```

Figure 15.3: `snark-verifier/src/loader/halo2/shim.rs#261–266`

The `<BaseFieldEccChip<C> as LimbsEncodingInstructions<C,LIMBS,BITS>>::assign_ec_point_from_limbs` function, shown in figure 15.4, does not check that the resulting `AssignedPoint` value is on the curve:

```
fn assign_ec_point_from_limbs(
    &self,
    ctx: &mut Self::Context,
    limbs: &[impl Deref<Target = Self::AssignedScalar>],
) -> Self::AssignedEcPoint {
    assert_eq!(limbs.len(), 2 * LIMBS);

    let ec_point = self.assign_point::<C>(
        ctx.main(0),
        ec_point_from_limbs::<_, LIMBS, BITS>(
            &limbs.iter().map(|limb| limb.value()).collect_vec(),
        ),
    );
```

```
    for (src, dst) in limbs
        .iter()

.zip_eq(iter::empty().chain(ec_point.x().limbs()).chain(ec_point.y().limbs()))
    {
        ctx.main(0).constrain_equal(src, dst);
    }

    ec_point
}
```

Figure 15.4: `snark-verifier/src/pcs/kzg/accumulator.rs#216–238`

Because of these issues, a malicious prover may be able to provide an off-curve point as an elliptic curve point witness, potentially causing elliptic curve arithmetic to misbehave and forging a proof.

Recommendations

Short term, either add an on-curve constraint to `load_private` or ensure that every use of it includes an on-curve check.

16. Native KZG accumulation decider accepts an empty vector	
Severity: **Medium**	Difficulty: **Low**
Type: Cryptography	Finding ID: TOB-AXIOM-16
Target: `snark-verifier/src/pcs/{kzg, ipa}/decider.rs`	

Description

Both the KZG and IPA native implementations of the `decide_all` function accept an empty vector of accumulators. This can allow an attacker to bypass verification by submitting an empty vector.

```
fn decide_all(
    dk: &Self::DecidingKey,
    accumulators: Vec<KzgAccumulator<M::G1Affine, NativeLoader>>,
) -> Result<(), Error> {
    accumulators
        .into_iter()
        .map(|accumulator| Self::decide(dk, accumulator))
        .try_collect::<_, Vec<_>, _>()?;
    Ok(())
}
```

Figure 16.1: `snark-verifier/src/pcs/kzg/decider.rs#L79-L89`

These implementations contrast with the EVM loader implementation of the function, which asserts that the accumulator vector is not empty:

```
fn decide_all(
    dk: &Self::DecidingKey,
    mut accumulators: Vec<KzgAccumulator<M::G1Affine, Rc<EvmLoader>>>,
) -> Result<(), Error> {
    assert!(!accumulators.is_empty());
```

Figure 16.2: `snark-verifier/src/pcs/kzg/decider.rs#L139-L143`

Exploit Scenario

An attacker is able to control the arguments to `decide_all` and passes an empty vector, causing the verification function to accept an invalid proof.

Recommendations

Short term, add an assertion that verifies that the vector is not empty.

Long term, add negative tests for verification and validation functions, ensuring that wrong or invalid arguments are not accepted.

17. Polynomial addition and subtraction assume polynomials have the same degree

Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-AXIOM-17
Target: `snark-verifier/src/util/poly.rs`	

Description

The `Polynomial` struct's addition and subtraction routines assume that the polynomials have the same degree. If one polynomial has a larger degree than the other, then the higher-degree terms will be silently truncated.

Figure 17.1 shows the implementation of the polynomial addition routine; if the polynomials do not have the same degree, coefficient truncation will occur because the resulting `zipped` iterator will stop as soon as one of the iterator arguments stops.

```
fn add(mut self, rhs: &'a Polynomial<F>) -> Polynomial<F> {
    parallelize(&mut self.0, |(lhs, start)| {
        for (lhs, rhs) in lhs.iter_mut().zip(rhs.0[start..].iter()) {
            *lhs += *rhs;
        }
    });
    self
}
```

Figure 17.1: `snark-verifier/src/util/poly.rs#L90-L98`

Figure 17.2 shows a failing test that checks whether $(1 + x) + (1 + x + x^2) == (2 + 2x + x^2)$:

```
   #[test]
    fn test_add_polynomials() {
        use crate::halo2_curves::bn256::Fr;
        use crate::util::poly::Polynomial;

        let a = Polynomial::new(vec![Fr::one(), Fr::one()]);
        let b = Polynomial::new(vec![Fr::one(), Fr::one(), Fr::one()]);
        let res = Polynomial::new(vec![Fr::one() + Fr::one(), Fr::one() + Fr::one(),
Fr::one()]);
        assert_eq!((a + &b).0, res.0);
    }
//     thread 'util::poly::tests::test_add_polynomials' panicked at 'assertion
failed: `(left == right)`
```

```
//     left: `[0x0000000000000000000000000000000000000000000000000000000000000002,
0x0000000000000000000000000000000000000000000000000000000000000002]`,
//    right: `[0x0000000000000000000000000000000000000000000000000000000000000002,
0x0000000000000000000000000000000000000000000000000000000000000002,
0x0000000000000000000000000000000000000000000000000000000000000001]`
```

Figure 17.2: Unit test for polynomial addition

The severity of this finding is marked as informational because the `Polynomial` structure appears to be used only for the IPA polynomial commitment scheme, which was marked as out of scope for this assessment.

Recommendations

Short term, compute the polynomial taking into account the full coefficients list; add unit tests for all operations; document the data structure with respect to the coefficient order.

Long term, investigate and add tests to all uses of the `zip` iterator, as they are a common source of issues.

18. FpChip::enforce_less_than_p incorrectly allows certain values above 2^t

Severity: **Informational**	Difficulty: **N/A**
Type: Data Validation	Finding ID: TOB-AXIOM-18
Target: `halo2-ecc/src/fields/fp.rs`	

Description

The `FpChip::enforce_less_than_p` function is used to check that field elements are in canonical form, but it allows certain non-canonical `CRTIntegers`.

Within `halo2_ecc`, the `CRTInteger` type represents integers $n \in [0, 2^t p)$, where p is the native field size, by splitting them into a native field element `native`, and a t-bit `OverflowInteger`, represented by the field `truncation`. The integer n is uniquely determined by the equations $n \equiv truncation \; mod \; 2^t$ and $n \equiv native \; mod \; p$.

Within the `FpChip` component, each element x of the non-native field F_q is then represented by a `CRTInteger`, where $2^t > q$ and $2^t p > q^2$, which can take on values above q. When checking equations on field elements, it is important to guarantee that they are reduced modulo q first, which is done via the `FpChip::enforce_less_than_p` function, as shown in figure 18.1:

```
// assuming `a` has been range checked to be a proper BigInt
// constrain the witness `a` to be `< p`
// then check if `a` is 0
fn is_zero(&self, ctx: &mut Context<F>, a: &CRTInteger<F>) -> AssignedValue<F> {
    self.enforce_less_than_p(ctx, a);
    // just check truncated limbs are all 0 since they determine the native value
    big_is_zero::positive::<F>(self.gate(), ctx, &a.truncation)
}
```

Figure 18.1: `halo2-ecc/src/fields/fp.rs#346–353`

However, figure 18.2 shows that `FpChip::enforce_less_than_p` checks only the `truncation` part of its argument:

```
    pub fn enforce_less_than_p(&self, ctx: &mut Context<F>, a: &CRTInteger<F>) {
        // a < p iff a - p has underflow
        let mut borrow: Option<AssignedValue<F>> = None;
        for (&p_limb, &a_limb) in self.p_limbs.iter().zip(a.truncation.limbs.iter())
{
            let lt = match borrow {
```

```
                None => self.range.is_less_than(ctx, a_limb, Constant(p_limb),
self.limb_bits),
                Some(borrow) => {
                    let plus_borrow = self.range.gate.add(ctx, Constant(p_limb),
borrow);
                    self.range.is_less_than(
                        ctx,
                        Existing(a_limb),
                        Existing(plus_borrow),
                        self.limb_bits,
                    )
                }
            };
            borrow = Some(lt);
        }
        self.range.gate.assert_is_const(ctx, &borrow.unwrap(), &F::one());
    }
}
```

Figure 18.2: `halo2-ecc/src/fields/fp.rs#80–100`

Certain `CRTIntegers` that represent values above q can still pass this check. Any value $n \geq 2^t$ of the form $a2^t + b$, where $a \geq 1$ and $0 \leq b < q$ will pass this check; the `truncation` field will be equal to b, and `native` will be equal to $a2^t + b \bmod p$.

This is illustrated in the `test_fp_eq` test case, shown in figure 18.3. In this test, a correctly reduced `CRTInteger` a is asserted to be equal to a maliciously constructed `CRTInteger` `a_evil`. A reduced version of `a_evil`, `a_evil_reduced`, is constructed via the `carry_mod` function and asserted to be not equal to a. Since all three `CRTIntegers` pass `enforce_less_than_p`, one would assume that reduction cannot change the result of an equality comparison.

```
fn load_private_evil<F: PrimeField, Fp: PrimeField>(chip: &FpChip<F,Fp>, ctx: &mut
Context<F>, a: BigInt) -> CRTInteger<F> {

    let a_vec = decompose_bigint::<F>(&(&a), chip.num_limbs, chip.limb_bits);
    let limbs = ctx.assign_witnesses(a_vec);

    let a_native = OverflowInteger::<F>::evaluate(
        &chip.range.gate,
        ctx,
        limbs.iter().copied(),
        chip.limb_bases.iter().copied(),
    );

    let a_native = chip.range.gate.add(ctx, QuantumCell::Existing(a_native),
QuantumCell::<F>::Constant((F::one()+F::one()).pow_vartime([(chip.num_limbs*chip.lim
b_bits) as u64])));

    let a_loaded =
```

```
        CRTInteger::construct(OverflowInteger::construct(limbs, chip.limb_bits),
a_native, a + (BigInt::one() << (chip.num_limbs * chip.limb_bits)));

    a_loaded
}

fn fp_eq_test<F: PrimeField>(
    ctx: &mut Context<F>,
    lookup_bits: usize,
    limb_bits: usize,
    num_limbs: usize,
    _a: Fq,
) {
    std::env::set_var("LOOKUP_BITS", lookup_bits.to_string());
    let range = RangeChip::<F>::default(lookup_bits);
    let chip = FpChip::<F, Fq>::new(&range, limb_bits, num_limbs);

    let a_witness = FpChip::<F, Fq>::fe_to_witness(&_a);
    let a = chip.load_private(ctx, a_witness.clone());
    chip.enforce_less_than_p(ctx,&a);

    let a_evil = load_private_evil(&chip, ctx, a_witness);
    chip.enforce_less_than_p(ctx,&a_evil);
    let is_equal0 = chip.is_equal_unenforced(ctx,&a,&a_evil);
    range.gate.assert_is_const(ctx, &is_equal0, &F::one());

    let a_evil_reduced = chip.carry_mod(ctx,&a_evil);

    chip.enforce_less_than_p(ctx,&a_evil_reduced);
    let is_equal1 = chip.is_equal_unenforced(ctx,&a,&a_evil_reduced);
    range.gate.assert_is_const(ctx, &is_equal1, &F::zero());
}

#[test]
fn test_fp_eq() {
    let k = K;
    let a = Fq::random(OsRng);

    let mut builder = GateThreadBuilder::<Fr>::mock();
    fp_eq_test(builder.main(0), k - 1, 88, 3, a);

    builder.config(k, Some(10));
    let circuit = RangeCircuitBuilder::<_, ZK>::mock(builder);

    MockProver::run::<_, ZK>(k as u32, &circuit,
vec![]).unwrap().assert_satisfied();
}
```

Figure 18.3: A test case illustrating this issue

In practice, this issue does not appear to be directly exploitable, since `CRTIntegers` are generally introduced to the circuit via `FpChip::load_private`. By constructing `native` from `truncation`, the `FpChip::load_private` function guarantees that all values are below 2^t:

```
fn load_private(&self, ctx: &mut Context<F>, a: BigInt) -> CRTInteger<F> {
    let a_vec = decompose_bigint::<F>(&a, self.num_limbs, self.limb_bits);
    let limbs = ctx.assign_witnesses(a_vec);

    let a_native = OverflowInteger::<F>::evaluate(
        self.range.gate(),
        ctx,
        limbs.iter().copied(),
        self.limb_bases.iter().copied(),
    );

    let a_loaded =
        CRTInteger::construct(OverflowInteger::construct(limbs, self.limb_bits),
a_native, a);

    // TODO: this range check prevents loading witnesses that are not in "proper"
representation form, is that ok?
    self.range_check(ctx, &a_loaded, Self::PRIME_FIELD_NUM_BITS as usize);
    a_loaded
}
```

Figure 18.4: The native field is constructed from a t-bit value.
(`halo2-ecc/src/fields/fp.rs#144–161`)

Recommendations

Short term, add a check to `FpChip::enforce_less_than_p` to enforce that `native` == `truncation` % q.

19. FpChip::assert_equal does not assert equality	
Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-AXIOM-19
Target: `halo2-ecc/src/fields/fp.rs`	

Description

The implementation of the `FpChip::assert_equal` function improperly compares its input a against itself, instead of against b, making the result of the comparison trivially true:

```
// assuming `a, b` have been range checked to be a proper BigInt
// constrain the witnesses `a, b` to be `< p`
// then assert `a == b` as BigInts
fn assert_equal(&self, ctx: &mut Context<F>, a: &Self::FieldPoint, b:
&Self::FieldPoint) {
    self.enforce_less_than_p(ctx, a);
    self.enforce_less_than_p(ctx, b);
    // a.native and b.native are derived from `a.truncation, b.truncation`, so no
need to check if they're equal
    for (limb_a, limb_b) in a.truncation.limbs.iter().zip(a.truncation.limbs.iter())
{
        ctx.constrain_equal(limb_a, limb_b);
    }
}
```

Figure 19.1: `a.truncation` is zipped with itself. (`halo2-ecc/src/fields/fp.rs#364–374`)

This function is the basis for several other equality functions, including the following:

- `Fp2Chip::assert_equal`
- `Fp12Chip::assert_equal`
- `EccChip::assert_equal`

`EccChip::assert_equal` is used to implement the `EcPointLoader::ec_point_assert_eq` function, as shown in figure 19.2:

```rust
fn ec_point_assert_eq(
    &self,
    _annotation: &str,
    lhs: &EcPoint<C, EccChip>,
    rhs: &EcPoint<C, EccChip>,
) {
    if let (Value::Constant(lhs), Value::Constant(rhs)) =
        (lhs.value().deref(), rhs.value().deref())
    {
        assert_eq!(lhs, rhs);
    } else {
        let lhs = lhs.assigned();
        let rhs = rhs.assigned();
        self.ecc_chip().assert_equal(&mut self.ctx_mut(), lhs.deref(), rhs.deref());
    }
}
```

Figure 19.2: `snark-verifier/src/loader/halo2/loader.rs#518–533`

This function is in turn used to implement the critical final check of the `Ipa::succinct_verify` function, shown in figure 19.3:

```rust
pub fn succinct_verify<L: Loader<C>>(
    svk: &IpaSuccinctVerifyingKey<C>,
    commitment: &Msm<C, L>,
    z: &L::LoadedScalar,
    eval: &L::LoadedScalar,
    proof: &IpaProof<C, L>,
) -> Result<IpaAccumulator<C, L>, Error> {
    let loader = z.loader();
    let h = loader.ec_point_load_const(&svk.h);
    let s = svk.s.as_ref().map(|s| loader.ec_point_load_const(s));
    let h = Msm::<C, L>::base(&h);

    let h_prime = h * &proof.xi_0;
    let lhs = {
…
    };
    let rhs = {
        let u = Msm::<C, L>::base(&proof.u);
        let v_prime = h_eval(&proof.xi(), z) * &proof.c;
        (u * &proof.c + h_prime * &v_prime).evaluate(None)
    };

    loader.ec_point_assert_eq("C_k == c[U] + v'[H']", &lhs, &rhs);

    Ok(IpaAccumulator::new(proof.xi(), proof.u.clone()))
}
```

Figure 19.3: `snark-verifier/src/pcs/ipa.rs#137–180`

The incorrect comparison in `FpChip::assert_equal` could allow malicious provers to forge proofs.

Exploit Scenario

A malicious prover generates a correctly formatted proof for an arbitrary `Ipa` statement and provides it to `Ipa::succinct_verify`. The final `assert_equal` is trivially satisfied, and the proof is accepted.

Recommendations

Short term, correct the implementation of `FpChip::assert_equal`.

Long term, implement a more extensive testing framework, with an emphasis on including both positive (should-pass) and negative (should-fail) tests for any functions used in security-critical components such as `Ipa::succinct_verify`. Negative testing of any of these `assert_equal` functions would have immediately triggered this bug.

20. Scalar rotation misbehaves on i32::MIN	
Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-AXIOM-20
Target: `snark-verifier/src/util/arithmetic.rs`	

Description

The `rotate_scalar` routine misbehaves when the `rotation` argument is `Rotation(i32::MIN)`. In debug mode, it will panic due to an overflow, while in release mode, it will compute an incorrect rotation ($g^{-18446744071562067968}$ instead of $g^{-2147483648}$).

This behavior is caused by the negation of the `i32 rotation` value and the subsequent cast of the value to `u64` that happen when the `rotation` value is negative:

```
/// Rotate an element to given `rotation`.
pub fn rotate_scalar(&self, scalar: F, rotation: Rotation) -> F {
    match rotation.0.cmp(&0) {
        Ordering::Equal => scalar,
        Ordering::Greater => scalar * self.gen.pow_vartime([rotation.0 as u64]),
        Ordering::Less => scalar * self.gen_inv.pow_vartime([(-rotation.0) as u64]),
    }
}
```

Figure 20.1: `snark-verifier/src/util/arithmetic.rs#L148-L156`

Because `-i32::MIN == i32::MIN` in release mode, the subsequent cast to `u64` returns $2^{64}-2^{31}$, which is not the desired value of 2^{31}.

In debug mode, this code will panic when receiving the `i32::MIN` value.

Recommendations

Short term, have the code cast the `i32 rotation` value to a larger type before computing the negation.

21. Several functions assume that arguments are non-empty

Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-AXIOM-21
Target: Several files	

Description

We identified several functions across the codebase that assume that their array arguments have at least one element. If this is not the case, the program will halt with a runtime error.

For example, figure 21.1 shows a function that can result in a runtime error in two ways: when `pairs` is empty and when `pairs` contains only tuples of `(scalar, base)`, where `base` is the constant point at infinity. In the second case, even if the function verifies that `fixed_base` is not empty, a runtime error will occur when the array is filtered and becomes empty (figure 21.2). This empty array will cause the implementation of `msm` to panic (figure 21.3).

```
fn multi_scalar_multiplication(
    pairs: &[(&<Self as ScalarLoader<C::Scalar>>::LoadedScalar, &EcPoint<'a, C,
EccChip>)],
) -> EcPoint<'a, C, EccChip> {
    let loader = &pairs[0].0.loader;

    let (constant, fixed_base, variable_base_non_scaled, variable_base_scaled) =
        pairs.iter().cloned().fold(
            (C::identity(), Vec::new(), Vec::new(), Vec::new()),
            |(
                mut constant,
                mut fixed_base,
                mut variable_base_non_scaled,
                mut variable_base_scaled,
            ),
             (scalar, base)| {
                match (scalar.value().deref(), base.value().deref()) {
                    (Value::Constant(scalar), Value::Constant(base)) => {
                        constant = (*base * scalar + constant).into()
                    }
                    (Value::Assigned(_), Value::Constant(base)) => {
                        fixed_base.push((scalar, *base))
                    }
    // ...
    let fixed_base_msm = (!fixed_base.is_empty())
        .then(|| {
```

```
            let fixed_base = fixed_base
                .into_iter()
                .map(|(scalar, base)| (scalar.assigned(), base))
                .collect_vec();
            loader
                .ecc_chip
                .borrow_mut()
                .fixed_base_msm(&mut loader.ctx_mut(), &fixed_base)
                .unwrap()
        })
```

Figure 21.1: snark-verifier/src/loader/halo2/loader.rs#L589-L633

```
fn fixed_base_msm(
    &mut self,
    ctx: &mut Self::Context,
    pairs: &[(impl Deref<Target = Self::AssignedScalar>, C)],
) -> Result<Self::AssignedEcPoint, Error> {
    let (scalars, points): (Vec<_>, Vec<_>) = pairs
        .iter()
        .filter_map(|(scalar, point)| {
            if point.is_identity().into() {
                None
            } else {
                Some((vec![scalar.deref().clone()], *point))
            }
        })
        .unzip();

    Ok(BaseFieldEccChip::<C>::fixed_base_msm::<C>(
        self,
        ctx,
        &points,
        &scalars,
        C::Scalar::NUM_BITS as usize,
        0,
        4,
    ))
}
```

Figure 21.2: snark-verifier/src/loader/halo2/shim.rs#L353-L378

```
// basically just adding up individual fixed_base::scalar_multiply except that we do
all batched normalization of cached points at once to further save inversion time
during witness generation
// we also use the random accumulator for some extra efficiency (which also works in
scalar multiply case but that is TODO)
pub fn msm<'v, F, FC, C>(
    chip: &EccChip<F, FC>,
    ctx: &mut Context<'v, F>,
    points: &[C],
    scalars: &[Vec<AssignedValue<'v, F>>],
```

```
    max_scalar_bits_per_cell: usize,
    window_bits: usize,
) -> EcPoint<F, FC::FieldPoint<'v>>
where
    F: PrimeField,
    C: CurveAffineExt,
    C::Base: PrimeField,
    FC: PrimeFieldChip<F, FieldType = C::Base, FieldPoint<'v> = CRTInteger<'v, F>>
        + Selectable<F, Point<'v> = FC::FieldPoint<'v>>,
{
    assert!((max_scalar_bits_per_cell as u32) <= F::NUM_BITS);
    let scalar_len = scalars[0].len();
```

Figure 21.3: `halo2-ecc/src/ecc/fixed_base.rs#L186-L204`

The following functions will result in a runtime error when provided with empty arguments:

- `bits_to_indicator` when provided with an empty `bits` argument (`halo2-lib/halo2-base/src/gates/flex_gate.rs#L429-L442`)
- `lagrange_and_eval` when provided with an empty `coords` argument (`halo2-lib/halo2-base/src/gates/flex_gate.rs#L601-L614`)
- `multi_scalar_multiplication` when provided with an empty `pairs` argument (`snark-verifier/src/loader/halo2/loader.rs#L589-L592`)
- `multi_scalar_multiplication` when provided with an empty `pairs` argument (`snark-verifier/src/loader/evm/loader.rs#L677-L690`)
- `aggregate` when provided with an empty `snarks` argument (`snark-verifier-sdk/src/halo2/aggregation.rs#L76-L136`)
    - The empty `snarks` argument would cause the `accumulators` argument to be empty and cause the system to crash on the call to `accumulators.pop().unwrap()`.
- `hash_tree_root` when provided with an empty `leaves` argument (`axiom-eth/src/util/mod.rs#L196-L197`)

Exploit Scenario

An API user calls `aggregate` without checking that the array is empty, causing the program to halt with a runtime error.

Recommendations

Short term, add checks for empty arrays to the affected functions, and have any functions that are not expected to handle empty arrays return errors when receiving such input. Add documentation for public functions stating their preconditions. Add tests that exercise these functions with empty arrays.

22. EVM verifier does not validate the deployment code

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-AXIOM-22
Target: `snark-verifier-sdk/src/evm.rs`, `snark-verifier/src/loader/evm/util.rs`	

Description

The `evm_verify` function does not validate the deployment code, and it accepts any instances and proofs when the deployment code is an empty vector. An empty or truncated deployment code can be caused by the function responsible for building the Yul code into EVM bytecode.

Figure 22.1 shows the `evm_verify` function and a successful test that accepts a fake proof when the deployment code is empty.

```
pub fn evm_verify(deployment_code: Vec<u8>, instances: Vec<Vec<Fr>>, proof: Vec<u8>)
{
    let calldata = encode_calldata(&instances, &proof);
    let success = {
        let mut evm =
ExecutorBuilder::default().with_gas_limit(u64::MAX.into()).build();

        let caller = Address::from_low_u64_be(0xfe);
        let verifier = evm.deploy(caller, deployment_code.into(),
0.into()).address.unwrap();
        let result = evm.call_raw(caller, verifier, calldata.into(), 0.into());

        dbg!(result.gas_used);

        !result.reverted
    };
    assert!(success);
}

#[test]
fn test_empty_deployment_code() {
    let circuit = StandardPlonk::rand(OsRng);
    evm_verify(vec![], circuit.instances(), b"fake proof".to_vec());
}
```

Figure 22.1: `snark-verifier-sdk/src/evm.rs#L179-L193`

An empty or truncated deployment code could be caused by the `compile_yul` and `split_by_ascii_whitespace` functions:

- After the compiler is called, the output is split by whitespace and decoded.
- The `split_by_ascii_whitespace` function is not correctly implemented, so it returns an empty vector when there is no whitespace. If there are whitespaces, but the string does not end with a whitespace, it will truncate the last chunk of the output.

Figure 22.2 shows two tests that trigger these issues on the `split_by_ascii_whitespace` function.

```
#[test]
fn test_split_by_ascii_whitespace_1() {
    let bytes = b" \x01 \x02   \x03";
    let split = split_by_ascii_whitespace(bytes);
    assert_eq!(split, [b"\x01", b"\x02", b"\x03"]);
}
// thread 'loader::evm::util::test_split_by_ascii_whitespace_1' panicked at
'assertion failed: `(left == right)`
//   left: `[[1], [2]]`,
//  right: `[[1], [2], [3]]`', snark-verifier/src/loader/evm/util.rs:162:5


#[test]
fn test_split_by_ascii_whitespace_2() {
    let bytes = b"123456789abc";
    let split = split_by_ascii_whitespace(bytes);
    assert_eq!(split, [b"123456789abc"]);
}
// thread 'loader::evm::util::test_split_by_ascii_whitespace_2' panicked at
'assertion failed: `(left == right)`
//   left: `[]`,
//  right: `[[49, 50, 51, 52, 53, 54, 55, 56, 57, 97, 98, 99]]`'
```

Figure 22.1: Two tests for the `split_by_ascii_whitespace` function

The Axiom team has stated that this finding has no impact because this is not the mechanism used to deploy code.

Exploit Scenario

The `solc` compiler changes the output format of the compilation command so that it does not return the last newline, causing the `split_by_ascii_whitespace` function to silently truncate the deployment code and trivial proof forgeries to be accepted.

Recommendations

Short term, add validations to the code returned by the `compile_yul` function and fix the implementation issues in the `split_by_ascii_whitespace` function.

Long term, pin and validate the `solc` binary to ensure that the binary has not been tampered with.

23. Values from load_random_point are used without strict checks	
Severity: **Informational**	Difficulty: **N/A**
Type: Data Validation	Finding ID: TOB-AXIOM-23
Target: `halo2-ecc/src/ecc/{mod,pippenger,fixed_base_pippenger}.rs`	

Description

The `ecc::load_random_point` function is used in certain `halo2-ecc` circuits to offset elliptic curve arithmetic operations by a prover-selected value. Once offset, arithmetic operations that might need to do point doubling in some cases can be calculated exclusively using the non-equal point addition formula. Because the point is prover-selected, the elliptic curve operations must be used in "strict" mode, which forbids points with equal x-coordinates. However, some elliptic curve operations are done in non-strict mode, which could allow a prover to perform proof forgery if the circuit is instantiated with certain elliptic curves.

As shown in figure 23.1, the `pippenger::multi_product` function uses a non-strict subtraction operation to calculate $[2^k - 1]rand\_base$ for various values of k. The `pippenger::multi_exp`, `pippenger::multi_exp_par`, and `fixed_base_pippenger::multi_product` functions also use this non-strict operation.

```
// we have acc[j] = G'[j] + (2^num_rounds - 1) * rand_base
rand_point = ec_double(chip, ctx, &rand_point);
rand_point = ec_sub_unequal(chip, ctx, &rand_point, &rand_base, false);
```

Figure 23.1: The non-strict `ec_sub_unequal` operation used in `pippenger::multi_product` (`halo2-ecc/src/ecc/pippenger.rs#140–143`)

This subtraction operation will trigger the issue described in finding TOB-AXIOM-10 if $rand\_point = \pm\ rand\_base$. Since $rand\_point\ = [2^k]rand\_base$, that equality occurs exactly when the order of $rand\_base$ is a factor of either $2^k - 1$ or $2^k + 1$. If the curve order is prime, it suffices to require that the order of the group is not $2^k \pm 1$, as noted by the comment "`assume 2^scalar_bits != +-1 mod modulus::<F>()`" in `multi_exp`. If this code is instantiated with a composite-order curve, `load_random_point` allows the prover to trigger this issue without violating that assumption.

As shown in figure 23.2 `fixed_base_pippenger::multi_exp` calls `ec_add_unequal` in a different way. For a given radix r, `ec_add_unequal` is called in the k-th iteration with

$\left[\sum_{i=1}^{k} 2^{ri}\right] rand\_point$ and $rand\_point$, which is problematic if $rand\_point$ has order dividing $\left(\sum_{i=1}^{k} 2^{ri}\right) \pm 1$ for any k in `[1,agg.len())`.

```
// compute sum_{k=0..t} agg[k] * 2^{radix * k} - (sum_k 2^{radix * k}) * rand_point
// (sum_{k=0..t} 2^{radix * k}) * rand_point = (2^{radix * t} - 1)/(2^radix - 1)
let mut sum = agg.pop().unwrap();
let mut rand_sum = rand_point.clone();
for g in agg.iter().rev() {
    for _ in 0..radix {
        sum = ec_double(chip, ctx, &sum);
        rand_sum = ec_double(chip, ctx, &rand_sum);
    }
    sum = ec_add_unequal(chip, ctx, &sum, g, true);
    chip.enforce_less_than(ctx, sum.x());

    if radix != 1 {
        // Can use non-strict as long as some property of the prime is true?
        rand_sum = ec_add_unequal(chip, ctx, &rand_sum, &rand_point, false);
    }
}
```

Figure 23.2: The non-strict `ec_add_unequal` operation used in `fixed_base_pippenger::multi_exp` (`halo2-ecc/src/ecc/fixed_base_pippenger.rs#234–250`)

Exploit Scenario

A developer uses these functions with a non-prime-order curve, such as E2 of `BN254`. A malicious prover then chooses a low-order random point, triggering the issue described in TOB-AXIOM-10 inside `ec_add_unequal` and forging a proof with an incorrect result.

Recommendations

Short term, use `ec_sub_unequal` and `ec_add_unequal` in strict mode in the `pippenger::multi_product`, `pippenger::multi_exp`, `pippenger::multi_exp_par`, `fixed_base_pippenger::multi_product`, and `fixed_base_pippenger::multi_exp` functions. Consider enforcing curve order assumptions at circuit construction time with explicit `assert!(...)` calls.

24. query_cell_at_pos assumes that the column index is valid

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-AXIOM- 24
Target: `zkevm-keccak/src/keccak_packed_multi.rs`	

Description

The `query_cell_at_pos` function returns the cell at a given position. If this position exists, it will provide the cell for the existing advice column. However, if the column index is larger than the number of existing columns, it will create only one new `CellColumn`, assuming that the column index is never larger than `self.columns.len()`.

```
pub(crate) fn query_cell_at_pos(
    &mut self,
    meta: &mut ConstraintSystem<F>,
    row_idx: i32,
    column_idx: usize,
) -> Cell<F> {
    let column = if column_idx < self.columns.len() {
        self.columns[column_idx].advice
    } else {
        let advice = meta.advice_column();
        let mut expr = 0.expr();
        meta.create_gate("Query column", |meta| {
            expr = meta.query_advice(advice, Rotation::cur());
            vec![0.expr()]
        });
        self.columns.push(CellColumn { advice, expr });
        advice
    };

    let mut cells = Vec::new();
    meta.create_gate("Query cell", |meta| {
        cells.push(Cell::new(meta, column, column_idx, row_idx));
        vec![0.expr()]
    });
    cells[0].clone()
}
```

Figure 24.1: `zkevm-keccak/src/keccak_packed_multi.rs#L279-L304`

Recommendations

Short term, have the code either verify that the column index equals the length of `self.columns` or create as many columns as necessary to ensure that `self.columns` reaches the desired number of cells.

25. Unchecked uses of zip could bypass checks on parse_account_proof_phase0 and parse_storage_proof_phase0

Severity: **Undetermined**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-AXIOM- 25
Target: `axiom-eth/src/storage/mod.rs`	

Description

The `parse_account_proof_phase0` function parses and performs validations on the Merkle Patricia Trie (MPT) proof, including a validation to ensure that the MPT root is the state root. However, the validation performed on the length of the `state_root_bytes` array checks only that it is below a maximum length; this means that an empty `state_root_bytes` array would bypass the equality constraints imposed therein:

```
impl<'chip, F: Field> EthStorageChip<F> for EthChip<'chip, F> {
    fn parse_account_proof_phase0(
        &self,
        ctx: &mut Context<F>,
        keccak: &mut KeccakChip<F>,
        state_root_bytes: &[AssignedValue<F>],
        addr: AssignedBytes<F>,
        proof: MPTFixedKeyProof<F>,
    ) -> EthAccountTraceWitness<F> {
        assert_eq!(32, proof.key_bytes.len());

        // check key is keccak(addr)
        assert_eq!(addr.len(), 20);
        let hash_query_idx = keccak.keccak_fixed_len(ctx, self.gate(), addr, None);
        let hash_bytes = &keccak.fixed_len_queries[hash_query_idx].output_assigned;

        for (hash, key) in hash_bytes.iter().zip(proof.key_bytes.iter()) {
            ctx.constrain_equal(hash, key);
        }

        // check MPT root is state root
        for (pf_root, root) in
proof.root_hash_bytes.iter().zip(state_root_bytes.iter()) {
            ctx.constrain_equal(pf_root, root);
        }
```

Figure 25.1: `axiom-eth/src/storage/mod.rs#L166-L189`

In fact, any `state_root_bytes` array that is a prefix of the `root_hash_bytes` array would bypass the checks because the function does not enforce that `proof.root_hash_bytes` and `state_root_bytes` have the same length.

A similar issue is present in the `parse_storage_proof_phase0` function:

```
fn parse_storage_proof_phase0(
    &self,
    ctx: &mut Context<F>,
    keccak: &mut KeccakChip<F>,
    storage_root_bytes: &[AssignedValue<F>],
    slot: AssignedBytes<F>,
    proof: MPTFixedKeyProof<F>,
) -> EthStorageTraceWitness<F> {
    assert_eq!(32, proof.key_bytes.len());

    // check key is keccak(slot)
    let hash_query_idx = keccak.keccak_fixed_len(ctx, self.gate(), slot, None);
    let hash_bytes = &keccak.fixed_len_queries[hash_query_idx].output_assigned;

    for (hash, key) in hash_bytes.iter().zip(proof.key_bytes.iter()) {
        ctx.constrain_equal(hash, key);
    }
    // check MPT root is storage_root
    for (pf_root, root) in
proof.root_hash_bytes.iter().zip(storage_root_bytes.iter()) {
        ctx.constrain_equal(pf_root, root);
    }
```

Figure 25.2: `axiom-eth/src/storage/mod.rs#L222-L242`

Exploit Scenario
An attacker is able to provide smaller byte arrays that bypass equality constraints in the proof parsing code.

Recommendations
Short term, use `zip_eq` instead of `zip` in both of the affected functions to ensure that iterators have the same length, ensuring that at least one of them is always checked with the correct length.

Long term, investigate all uses of the `zip` operator for similar issues, and replace `zip` with `zip_eq` where iterators of unequal length are not necessary.

26. The hex_prefix_encode and hex_prefix_encode_first functions assume that the is_odd parameter is a bit	
Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-AXIOM-26
Target: `axiom-eth/src/mpt/mod.rs`	

Description

Both the `hex_prefix_encode` and `hex_prefix_encode_first` functions assume that the `is_odd` `AssignedValue` is either zero or one. The checks are made in the existing callsites, but nothing prevents these functions from being called with a value other than a bit. If the functions are called with an `is_odd` value that is not a bit, incorrect behavior would occur.

```
pub fn hex_prefix_encode_first<F: ScalarField>(
    ctx: &mut Context<F>,
    gate: &impl GateInstructions<F>,
    first_nibble: AssignedValue<F>,
    is_odd: AssignedValue<F>,
    is_ext: bool,
) -> AssignedValue<F> {
    let sixteen = gate.get_field_element(16);
    let thirty_two = gate.get_field_element(32);
    if is_ext {
        gate.inner_product(
            ctx,
            [Existing(is_odd), Existing(is_odd)],
            [Constant(sixteen), Existing(first_nibble)],
        )
    } else {
        // (1 - is_odd) * 32 + is_odd * (48 + x_0)
        // | 32 | 16 | is_odd | 32 + 16 * is_odd | is_odd | x_0 | out |
        let pre_val = thirty_two + sixteen * is_odd.value();
        let val = pre_val + *first_nibble.value() * is_odd.value();
        ctx.assign_region_last(
            [
                Constant(thirty_two),
                Constant(sixteen),
                Existing(is_odd),
                Witness(pre_val),
                Existing(is_odd),
                Existing(first_nibble),
                Witness(val),
            ],
```

```
            [0, 3],
        )
    }
}
```

Figure 26.1: `axiom-eth/src/mpt/mod.rs#L960-L993`

Recommendations

Short term, add documentation to the `hex_prefix_encode` and `hex_prefix_encode_first` functions stating that `is_odd` must have a bit value.

Long term, use the Rust type system to enforce this property at the compiler level.

27. batch_invert_and_mul ignores zero elements and panics on empty arrays

Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-AXIOM-27
Target: `snark-verifier/src/util/arithmetic.rs`	

Description

The `batch_invert_and_mul` function batch-inverts an array of elements (`values`) and multiplies it by a given coefficient. However, if the array contains zeros, the function will ignore them, leaving them unchanged. This behavior could cause unexpected issues, as the array would be successfully inverted even though some of its elements are not actually invertible.

```rust
/// Batch invert [`PrimeField`] elements and multiply all with given coefficient.
pub fn batch_invert_and_mul<F: PrimeField>(values: &mut [F], coeff: &F) {
    let products = values
        .iter()
        .filter(|value| !value.is_zero_vartime())
        .scan(F::one(), |acc, value| {
            *acc *= value;
            Some(*acc)
        })
        .collect_vec();

    let mut all_product_inv = products.last().unwrap().invert().unwrap() * coeff;

    for (value, product) in values
        .iter_mut()
        .rev()
        .filter(|value| !value.is_zero_vartime())
        .zip(products.into_iter().rev().skip(1).chain(Some(F::one())))
    {
        let mut inv = all_product_inv * product;
        mem::swap(value, &mut inv);
        all_product_inv *= inv;
    }
}
```

Figure 27.1: `snark-verifier/src/util/arithmetic.rs#L51-L73`

Additionally, if the `values` array is empty or contains only zeros, the function will panic when getting the last element of the `products` variable.

Exploit Scenario

An attacker passes zeros in the batch-inversion array, which are kept unchanged. Afterward, these values are used, assuming they are invertible, but the array actually contains non-invertible elements.

Recommendations

Short term, add checks to the inversion function to validate that the argument array is not empty; consider having the function error out when it encounters non-invertible elements.

28. Proof caching occurs before proof validation

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-AXIOM-28
Target: `snark-verifier-sdk/src/halo2.rs`	

Description

The proof generation function, `gen_proof`, caches the proof if a path is provided for it. However, it does so before verifying that the generated proof is valid. If, for some reason, the proof is invalid, the invalid proof would be cached and returned every time the `gen_proof` function is called again.

```
/// Caches the instances and proof if `path = Some(instance_path, proof_path)` is
specified.
pub fn gen_proof<'params, C, P, V>(
    // TODO: pass Option<&'params ParamsKZG<Bn256>> but hard to get lifetimes to
work with `Cow`
    params: &'params ParamsKZG<Bn256>,
    pk: &ProvingKey<G1Affine>,
    circuit: C,
    instances: Vec<Vec<Fr>>,
    path: Option<(&Path, &Path)>,
) -> Vec<u8>
where
    C: Circuit<Fr>,
    P: Prover<'params, KZGCommitmentScheme<Bn256>>,
    V: Verifier<
        'params,
        KZGCommitmentScheme<Bn256>,
        Guard = GuardKZG<'params, Bn256>,
        MSMAccumulator = DualMSM<'params, Bn256>,
    >,
{
    if let Some((instance_path, proof_path)) = path {
        let cached_instances = read_instances(instance_path);
        if matches!(cached_instances, Ok(tmp) if tmp == instances) &&
proof_path.exists() {
            #[cfg(feature = "display")]
            let read_time = start_timer!(|| format!("Reading proof from
{proof_path:?}"));

            let proof = fs::read(proof_path).unwrap();

            #[cfg(feature = "display")]
            end_timer!(read_time);
```

```
            return proof;
        }
    }

    let instances = instances.iter().map(Vec::as_slice).collect_vec();

    #[cfg(feature = "display")]
    let proof_time = start_timer!(|| "Create proof");

    let mut transcript =
        PoseidonTranscript::<NativeLoader, Vec<u8>>::from_spec(vec![],
POSEIDON_SPEC.clone());
    let rng = StdRng::from_entropy();
    create_proof::<_, P, _, _, _, _>(params, pk, &[circuit], &[&instances], rng,
&mut transcript)
        .unwrap();
    let proof = transcript.finalize();

    #[cfg(feature = "display")]
    end_timer!(proof_time);

    if let Some((instance_path, proof_path)) = path {
        write_instances(&instances, instance_path);
        fs::write(proof_path, &proof).unwrap();
    }

    debug_assert!({
        let mut transcript_read =
            PoseidonTranscript::<NativeLoader,
&[u8]>::new::<SECURE_MDS>(proof.as_slice());
        VerificationStrategy::<_, V>::finalize(
            verify_proof::<_, V, _, _, _>(
                params.verifier_params(),
                pk.get_vk(),
                AccumulatorStrategy::new(params.verifier_params()),
                &[instances.as_slice()],
                &mut transcript_read,
            )
            .unwrap(),
        )
    });

    proof
}
```

Figure 28.1: `snark-verifier-sdk/src/halo2.rs#L76-L145`

We also recommend converting the debug assertion to a regular assertion with `assert!`.

Recommendations

Short term, have the `gen_proof` function validate the generated proof before caching it to a file; convert the debug assertion to a regular assertion.

29. Merkle root computation does not differentiate leaf data hashing and inner node hashing

Severity: **Informational**	Difficulty: **High**
Type: Cryptography	Finding ID: TOB-AXIOM-29
Target: `axiom-eth/src/keccak/mod.rs`	

Description

The values hashed during the computation of a Merkle tree root should always be domain-separated and should use a different hash function from the one used to hash the leaf data (e.g., by prepending a prefix to distinguish the hashing of a leaf from the hashing of the concatenation of two inner nodes). In the `KeccakChip::merkle_tree_root` function, neither a prefix nor a domain separation is used to compute the hash of the inner nodes in the Merkle tree.

```
// bottom layer hashes
let mut hashes = leaves
    .chunks(2)
    .into_iter()
    .map(|pair| {
        let leaves_concat = [&pair[0][..], &pair[1][..]].concat();
        self.keccak_fixed_len(ctx, gate, leaves_concat, None)
    })
    .collect_vec();
```

Figure 29.1: `axiom-eth/src/keccak/mod.rs#L219-L241`

This implementation allows certain Merkle trees with different overall shapes to have the same root. For example, a single leaf with data `keccak(leaf1)||keccak(leaf2)` would lead to the same Merkle root as a tree with two leaves with data `leaf1` and `leaf2`, `keccak(keccak(leaf1)||keccak(leaf2))`. If the correct tree shape is not enforced while checking Merkle membership proofs against this root, incorrect proofs may be accepted.

Additionally, the computation of Merkle mountain ranges are also impacted by the lack of domain separation. When the number of leaves is a power of two, the example above applies similarly. Otherwise, if the smallest peak of the mountain range contains more than one leaf, it may be replaced by a peak with a single leaf.

Recommendations

Short term, add different domain-separated prefixes to the data to be hashed in the Merkle root computation, differentiating between the hash of a leaf and the hash of two inner nodes.

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories	
Category	Description
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

B. Code Maturity Categories

The following tables describe the code maturity categories and rating criteria used in this document.

Code Maturity Categories	
Category	Description
Arithmetic	The proper use of mathematical operations and semantics
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Cryptography and Key Management	The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution
Data Handling	The safe handling of user inputs and data processed by the system
Documentation	The presence of comprehensive and readable codebase documentation
Memory Safety and Error Handling	The presence of memory safety and robust error-handling mechanisms
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage

Rating Criteria	
Rating	**Description**
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.
Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.
Not Applicable	The category is not applicable to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. Automated Testing

This section describes the setup for the various automated analysis tools used during this audit.

cargo-audit

The Cargo plugin `cargo-audit` can be installed using the command `cargo install cargo-audit`. Invoking `cargo audit` in the root directory of the project runs the tool.

By running `cargo-audit` on the repositories under audit, we identified two vulnerable dependencies. However, none of the vulnerabilities appear to affect the codebase since the vulnerable functions are not used.

Clippy

The Rust linter Clippy can be installed using `rustup` by running the command `rustup component add clippy`. Invoking `cargo clippy` in the root directory of the project runs the tool.

We ran Clippy on the five repositories. This run did not identify any serious issues, but it found a few idiomatic issues in the `snark-verifier` codebase. We recommend adding a GitHub action on the repository that prevents code from being pushed with Clippy warnings.

D. Code Quality Findings

We identified the following code quality issues through manual and automatic code review.

- **The Keccak implementation uses a clone of a clone**, which is not necessary:

```
// Now calculate `a ^ ((~b) & c)` by doing `lookup[3 - 2*a + b - c]`
for i in 0..5 {
    let input = scatter::expr(3, part_size_base) - 2.expr() * input[i].clone()
        + input[(i + 1) % 5].clone()
        - input[(i + 2) % 5].clone().clone();
```

Figure D.1: `hashes/zkevm-keccak/src/keccak_packed_multi.rs#L1136-L1138`

- **The following mutable iterator can be immutable.** The `row` value is not changed, so using `.iter()` or having the code iterate over `&self.rows` would suffice:

```
pub(crate) fn start_region(&mut self) -> usize {
    // Make sure all rows start at the same column
    let width = self.get_width();
    #[cfg(debug_assertions)]
    for row in self.rows.iter_mut() {
        self.num_unused_cells += width - *row;
    }
    self.rows = vec![width; self.height];
    width
}
```

Figure D.2: `hashes/zkevm-keccak/src/keccak_packed_multi.rs#L336-L345`

- **Functions lack documentation with necessary conditions.** The `range_check` function requires certain bounds on `range_bits` (e.g., that `range_bits != 0`), but this is not documented and would cause the function to result in a runtime error.

```
fn range_check(&self, ctx: &mut Context<F>, a: AssignedValue<F>, range_bits:
usize) {
    // the number of limbs
    let k = (range_bits + self.lookup_bits - 1) / self.lookup_bits;
    // println!("range check {} bits {} len", range_bits, k);
    let rem_bits = range_bits % self.lookup_bits;

    debug_assert!(self.limb_bases.len() >= k);

    if k == 1 {
        ctx.cells_to_lookup.push(a);
    } else {
        let limbs = decompose_fe_to_u64_limbs(a.value(), k, self.lookup_bits)
            .into_iter()
            .map(|x| Witness(F::from(x)));
```

```
        let row_offset = ctx.advice.len() as isize;
        let acc = self.gate.inner_product(ctx, limbs,
self.limb_bases[..k].to_vec());
        // the inner product above must equal `a`
        ctx.constrain_equal(&a, &acc);
        // we fetch the cells to lookup by getting the indices where `limbs`
were assigned in `inner_product`. Because `limb_bases[0]` is 1, the
progression of indices is 0,1,4,...,4+3*i
        ctx.cells_to_lookup.push(ctx.get(row_offset));
        for i in 0..k - 1 {
            ctx.cells_to_lookup.push(ctx.get(row_offset + 1 + 3 * i as
isize));
        }
    };
```

Figure D.3: halo2-base/src/gates/range.rs#L395-L418

Similarly, the `is_less_than` function requires bounds on the `num_bits` argument, since a maliciously chosen value would cause an out-of-bounds access in the `self.gate.pow_of_two` vector.

```
fn is_less_than(
    &self,
    ctx: &mut Context<F>,
    a: impl Into<QuantumCell<F>>,
    b: impl Into<QuantumCell<F>>,
    num_bits: usize,
) -> AssignedValue<F> {
    let a = a.into();
    let b = b.into();

    let k = (num_bits + self.lookup_bits - 1) / self.lookup_bits;
    let padded_bits = k * self.lookup_bits;
    let pow_padded = self.gate.pow_of_two[padded_bits];
```

Figure D.4: halo2-base/src/gates/range.rs#L471-L483

- **There is a typo in the following gate label.** The `Vertical` gate label should read "1 column a + b * c = out".

```
   fn create_gate(&self, meta: &mut ConstraintSystem<F>) {
        meta.create_gate("1 column a * b + c = out", |meta| {
            let q = meta.query_selector(self.q_enable);

            let a = meta.query_advice(self.value, Rotation::cur());
            let b = meta.query_advice(self.value, Rotation::next());
            let c = meta.query_advice(self.value, Rotation(2));
            let out = meta.query_advice(self.value, Rotation(3));

            vec![q * (a + b * c - out)]
        })
```

```
    }
}
```

Figure D.5: halo2-base/src/gates/flex_gate.rs#L66-L78

- **There is an unnecessary range check in the `FieldChip::neg_divide` function.** This range check is redundant given the range check already performed by the `FieldChip::load_private` function, so it can be deleted.

```
let quot = self.load_private(ctx, Self::fe_to_witness(&quot_val));
self.range_check(ctx, &quot, Self::PRIME_FIELD_NUM_BITS as usize);

// constrain quot * b + a = 0 mod p
let quot_b = self.mul_no_carry(ctx, &quot, b);
let quot_constraint = self.add_no_carry(ctx, &quot_b, a);
self.check_carry_mod_to_zero(ctx, &quot_constraint);

quot
```

Figure D.6: halo2-ecc/src/fields/mod.rs#199–207

- **The `MockTranscript::common_scalar` function assumes it will be called only once.** If the behavior of the `VerifyingKey::hash_into` function changes in a future version of `halo2`, this would no longer correctly include the whole verifying key in the transcript. The `MockTranscript` component should contain an `Option<Scalar>` type, and this function should return `Err(...)` if the value of `self.0` is not None.

```
fn common_scalar(&mut self, scalar: C::Scalar) -> io::Result<()> {
    self.0 = scalar;
    Ok(())
}
```

Figure D.7: snark-verifier/src/system/halo2.rs#712–715

- **The following comment is incorrect and does not match the circuit's logic.** When the `prefix_parsed.is_big` value is set, the correct RLP field length is given by the `byte_value(len)` function; otherwise, it is given by the `prefix_parsed.next_len` function. The implementation logic selects the function correctly, but the comment indicates the reverse.

```
// * field_rlc.rlc_len = prefix_parsed.is_big * prefix_parsed.next_len
//                      + (1 - prefix_parsed.is_big) * byte_value(len)
```

Figure D.8: axiom-eth/src/rlp/mod.rs#325–326

- **Memory-consuming infinite loop will occur if `num_limbs == 0`.** If the `FpChip::new` function is called with a `num_limbs` parameter equal to `0`, this loop will continue inserting new elements into the `limb_bases` array indefinitely, consuming memory until the process is killed or an error occurs within the Vec component (e.g., allocation failure or overflow):

```
let mut limb_bases = Vec::with_capacity(num_limbs);
limb_bases.push(F::one());
while limb_bases.len() != num_limbs {
    limb_bases.push(limb_base * limb_bases.last().unwrap());
}
```

Figure D.9: `halo2-ecc/src/fields/fp.rs#57–61`

- **Redundant matches on the strategy since only `Vertical` is used.** The `FlexGateConfig::configure` and `BasicGateConfig::configure` functions match on the strategy, but all cases are handled in the same match arm.

```
impl<F: ScalarField> BasicGateConfig<F> {
    pub fn configure(meta: &mut ConstraintSystem<F>, strategy: GateStrategy,
phase: u8) -> Self {
        let value = match phase {
            0 => meta.advice_column_in(FirstPhase),
            1 => meta.advice_column_in(SecondPhase),
            2 => meta.advice_column_in(ThirdPhase),
            _ => panic!("Currently BasicGate only supports {MAX_PHASE}
phases"),
        };
        meta.enable_equality(value);

        let q_enable = meta.selector();

        match strategy {
            GateStrategy::Vertical => {
                let config = Self { q_enable, value, _marker: PhantomData };
                config.create_gate(meta);
                config
            }
        }
    }
```

Figure D.10: `halo2-base/src/gates/flex_gate.rs#L45-L64`

A similar case occurs in the `RangeConfig::configure` function:

```
impl<F: ScalarField> RangeConfig<F> {
    pub fn configure(
        meta: &mut ConstraintSystem<F>,
        range_strategy: RangeStrategy,
        num_advice: &[usize],
        num_lookup_advice: &[usize],
        num_fixed: usize,
        lookup_bits: usize,
```

```
        // params.k()
        circuit_degree: usize,
    ) -> Self {
        assert!(lookup_bits <= 28);
        let lookup = meta.lookup_table_column();

        let gate = FlexGateConfig::configure(
            meta,
            match range_strategy {
                RangeStrategy::Vertical => GateStrategy::Vertical,
            },
            num_advice,
            num_fixed,
            circuit_degree,
        );
```

Figure D.11: halo2-base/src/gates/range.rs#L49-L71

- **The `EvmTranscript` component implicitly assumes that all transcript inputs are scalars or curve points.** The `EvmTranscript::squeeze_challenge` function appends the byte `0x01` to its buffer if no other inputs have been added. If the one-byte sequence `[0x01]` is a valid input to the transcript, the empty input `[]` will have the same transcript result as `[0x01]`. Currently, one-byte sequences are not a possible input because the `Transcript` trait supports only the ability to write scalars and elliptic curve points. However, `EvmTranscript` does not document that only those inputs are allowed.

```
let len = if self.buf.len() == 0x20 {
    assert_eq!(self.loader.ptr(), self.buf.end());
    let buf_end = self.buf.end();
    let code = format!("mstore8({buf_end}, 1)");
    self.loader.code_mut().runtime_append(code);
    0x21
} else {
    self.buf.len()
};
let hash_ptr = self.loader.keccak256(self.buf.ptr(), len);
```

Figure D.12: snark-verifier/src/system/halo2/transcript/evm.rs#77–86

- **The following is an unnecessary conversion to u64.**

```
// returns little-endian bit vectors
fn num_to_bits(
    &self,
    ctx: &mut Context<F>,
    a: AssignedValue<F>,
    range_bits: usize,
) -> Vec<AssignedValue<F>> {
    let a_bytes = a.value().to_repr();
```

```
    let bits = a_bytes
        .as_ref()
        .iter()
        .flat_map(|byte| (0..8u32).map(|i| (*byte as u64 >> i) & 1))
```

Figure D.13: `halo2-base/src/gates/flex_gate.rs#L896-L906`

- **The following function is called `batch_invert` but it does not perform batch inversion.** Instead of batch inversion, the function computes the inverse of each input value.

```
/// Batch invert field elements.
fn batch_invert<'a>(values: impl IntoIterator<Item = &'a mut
Self::LoadedScalar>)
where
    Self::LoadedScalar: 'a,
{
    values
        .into_iter()
        .for_each(|value| *value =
LoadedScalar::invert(value).unwrap_or_else(|| value.clone()))
}
```

Figure D.14: `snark-verifier/src/loader.rs#L240-L249`

- **There is an unused file in the codebase.** The file `snark-verifier/src/system/halo2/aggregation.rs` is not used or built with the rest of the codebase.

- **The EVM loader implementations of the `ec_point_assert_eq` and `assert_eq` functions are empty**, meaning that if they were to be called, they would accept any input values. We recommend using the `unimplemented!` or `todo!` macros in such cases.

```
fn ec_point_assert_eq(&self, _: &str, _: &EcPoint, _: &EcPoint) {}
```

Figure D.15: `snark-verifier/src/loader/evm/loader.rs#L651`

- **The `compile` function lacks documentation.** The function at `snark-verifier/src/system/halo2.rs#L105-L127` should have documentation on why the `evaluations` iterator is missing the `polynomials.quotient_query()` function.

- **The string-based API is error-prone.** The `EthBlockHeaderTraceWitness` type defines a single `get` method to get individual decoded block header fields. To ensure that potential issues are caught early (at compile time, rather than runtime), we recommend redefining the type using one getter per field instead.

```
pub fn get(&self, header_field: &str) -> &RlpFieldWitness<F> {
    match header_field {
        "parent_hash" | "parentHash" => &self.rlp_witness.field_witness[0],
        "ommers_hash" | "ommersHash" => &self.rlp_witness.field_witness[1],
        "beneficiary" => &self.rlp_witness.field_witness[2],
        "state_root" | "stateRoot" => &self.rlp_witness.field_witness[3],
        "transactions_root" | "transactionsRoot" =>
            &self.rlp_witness.field_witness[4],
        "receipts_root" | "receiptsRoot" =>
            &self.rlp_witness.field_witness[5],
        "logs_bloom" | "logsBloom" => &self.rlp_witness.field_witness[6],
        "difficulty" => &self.rlp_witness.field_witness[7],
        "number" => &self.rlp_witness.field_witness[8],
        "gas_limit" | "gasLimit" => &self.rlp_witness.field_witness[9],
        "gas_used" | "gasUsed" => &self.rlp_witness.field_witness[10],
        "timestamp" => &self.rlp_witness.field_witness[11],
        "extra_data" | "extraData" => &self.rlp_witness.field_witness[12],
        "mix_hash" | "mixHash" => &self.rlp_witness.field_witness[13],
        "nonce" => &self.rlp_witness.field_witness[14],
        "basefee" => &self.rlp_witness.field_witness[15],
        _ => panic!("Invalid header field"),
    }
}
```

Figure D.16: `axiom-eth/src/block_header/mod.rs:#L108-L129`

- **An incorrect comment on the `KeccakChip::keccak_var_len` function does not match the circuit's logic.** The comment states that the function returns `(output_assigned, output_bytes)`, but it returns the index of the query in the `var_len_queries` vector. This comment should be updated.

- **The `limbs_be_to_u128` function in `axiom-eth/src/util/mod.rs` could panic.** The `limbs_be_to_u128` function takes as input an array of limbs and a `limb_bits` value, which determines the size of the limbs. To determine the number of chunks, the function divides 128 by the `limb_bits` value. The function expects that 128 is divisible by `limb_bits`, so it has an assertion that ensures that `128 % limb_bits == 0`. However, if the value of `limb_bits` is `0`, this will result in a division-by-zero panic. Consider handling this edge case more gracefully with a detailed error message.

- **The number of Merkle mountain range leaves could be constrained to be less than 2^{max_bits}.** The `EthBlockHeaderChainCircuit::create_circuit` method calls the `GateChip::num_to_bits` function to compute the bit representation of the number of leaf nodes in the Merkle mountain range. The call uses `self.max_depth + 1` as the maximum bit size, but the bit size could actually be constrained to be at most the value of `self.max_depth`.

- **Debug assertions in the `mpt/mod.rs` file should be assertions.** There are currently three instances of calls to `debug_assert_eq` in the `mpt/mod.rs` file. Each of these `debug_assert_eq` calls checks that the length of a certain object is the expected length. Elsewhere in this file, similar checks are performed with a regular `assert_eq` call instead. Consider replacing these three calls to `debug_assert_eq` with `assert_eq`.

E. Fix Review Results

When undertaking a fix review, Trail of Bits reviews the fixes implemented for issues identified in the original report. This work involves a review of specific areas of the source code and system configuration, not comprehensive analysis of the system.

Starting on June 1, 2023, Trail of Bits reviewed the fixes and mitigations implemented by the Axiom team for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue.

In summary, of the 29 issues described in this report, Axiom has resolved 25 issues, has partially resolved two issues, and has not resolved the remaining two issues. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Severity	Status
1	Incorrect limb decomposition due to bit-shifts larger than integer size	Low	Resolved
2	Risk of unconstrained inner product in release builds	Medium	Resolved
3	idx_to_indicator circuit is underconstrained	High	Resolved
4	ecdsa_verify_no_pubkey_check can fail on signatures from crafted public keys	Medium	Resolved
5	log2_ceil function miscomputes its result when x input is zero	Low	Resolved
6	GateChip::num_to_bits depends on implementation-specific details of the underlying field	Low	Resolved
7	RangeChip::get_last_bit returns the wrong value	Low	Resolved
8	Validations missing in release builds	Medium	Resolved

9	Keccak implementation cannot hash arbitrarily large inputs	Informational	Unresolved
10	Field division of zero by zero is unconstrained	Medium	Resolved
11	Incorrect point-at-infinity handling in elliptic curve operations	Medium	Resolved
12	FpChip::load_private allows non-reduced field elements	Informational	Resolved
13	scalar_multiply can return underconstrained results	High	Resolved
14	Witness may be underconstrained if two gates overlap with more than one cell	Informational	Unresolved
15	EccChip::load_private does not enforce that witness values are on-curve	Informational	Resolved
16	Native KZG accumulation decider accepts an empty vector	Medium	Resolved
17	Polynomial addition and subtraction assume polynomials have the same degree	Informational	Resolved
18	FpChip::enforce_less_than_p incorrectly allows certain values above 2^t	Informational	Resolved
19	FpChip::assert_equal does not assert equality	High	Resolved
20	Scalar rotation misbehaves on i32::MIN	Low	Resolved
21	Several functions assume that arguments are non-empty	Low	Resolved
22	EVM verifier does not validate the deployment code	Informational	Resolved

23	Values from load_random_point are used without strict checks	Informational	Partially Resolved
24	query_cell_at_pos assumes that the column index is valid	Informational	Resolved
25	Unchecked uses of zip could bypass checks on parse_account_proof_phase0 and parse_storage_proof_phase0	Undetermined	Resolved
26	The hex_prefix_encode and hex_prefix_encode_first functions assume that the is_odd parameter is a bit	Informational	Resolved
27	batch_invert_and_mul ignores zero elements and panics on empty arrays	Low	Resolved
28	Proof caching occurs before proof validation	Informational	Resolved
29	Merkle root computation does not differentiate leaf data hashing and inner node hashing	Informational	Partially Resolved

Detailed Fix Review Results

TOB-AXIOM-1: Incorrect limb decomposition due to bit-shifts larger than integer size
Resolved. The `decompose_biguint` now casts the mask to a `u64`. The `decompose_u64_digits_to_limbs` was fixed, but only with the addition of a `debug_assert`.

TOB-AXIOM-2: Risk of unconstrained inner product in release builds
Resolved. Each `debug_assert_eq` instance identified in finding TOB-AXIOM-2 has been replaced with `assert_eq`.

TOB-AXIOM-3: idx_to_indicator circuit is underconstrained
Resolved. The circuit has been updated to properly constrain the vector to have a zero in all positions except for position `idx`, where it is constrained to equal one.

TOB-AXIOM-4: ecdsa_verify_no_pubkey_check can fail on signatures from crafted public keys
Resolved. The circuit has been modified so that it accepts `u1*G == u2*PK` but forbids `u1*G == -(u2*PK)`.

TOB-AXIOM-5: log2_ceil function miscomputes its result when x input is zero
Resolved. The `log2_ceil` function has been updated to return zero when it receives a zero-value input.

TOB-AXIOM-6: GateChip::num_to_bits depends on implementation-specific details of the underlying field
Resolved. All of the instances of `to_repr` enumerated in finding TOB-AXIOM-6 have been addressed, except for one instance in `snark-verifier`. The Axiom team decided not to update the instance in `snark-verifier` because it would require trait changes throughout the codebase.

TOB-AXIOM-7: RangeChip::get_last_bit returns the wrong value
Resolved. The `RangeChip::get_last_bit` function has been updated to properly constrain and return the correct value.

TOB-AXIOM-8: Validations missing in release builds
Resolved. The examples enumerated in finding TOB-AXIOM-8 have been updated to include the proper validations.

TOB-AXIOM-9: Keccak implementation cannot hash arbitrarily large inputs
Unresolved. The Axiom team has not addressed this finding.

TOB-AXIOM-10: Field division of zero by zero is unconstrained
Resolved. The `divide` and `neg_divide` functions have been renamed to `divide_unsafe` and `neg_divide_unsafe` to indicate that zero checks are not performed. New `divide`

and `neg_divide` functions have been added that perform these checks and then call the unsafe functions. This will help ensure that the caller of these functions is aware of whether these checks are being performed.

TOB-AXIOM-11: Incorrect point-at-infinity handling in elliptic curve operations
Resolved. Documentation has been added to the affected functions identified in finding TOB-AXIOM-11.

TOB-AXIOM-12: FpChip::load_private allows non-reduced field elements
Resolved. The API has been changed to separate reduced and unreduced field elements. Instances of a `FieldChip` that require reduced field elements now must use the `FieldChip::ReducedFieldPoint` associated type, which is produced from `FieldChip::enforce_less_than` or `FieldChip::load_private_reduced`.

TOB-AXIOM-13: scalar_multiply can return underconstrained results
Resolved. The `scalar_multiply` function has been updated to take in a `scalar_is_safe` Boolean flag as input. This `scalar_is_safe` flag is passed to `ec_add_unequal` as the `is_strict` flag, rather than always using `false`.

TOB-AXIOM-14: Witness may be underconstrained if two gates overlap with more than one cell
Unresolved. The Axiom team has not addressed this finding.

TOB-AXIOM-15: EccChip::load_private does not enforce that witness values are on-curve
Resolved. The `load_private` and `assign_point` functions have been renamed to `load_private_unchecked` and `assign_point_unchecked` to indicate that on-curve checks are not performed. New `load_private` and `assign_point` functions have been added that perform these checks and then call the unchecked functions. This will help ensure that the caller of these functions is aware of whether these checks are being performed.

TOB-AXIOM-16: Native KZG accumulation decider accepts an empty vector
Resolved. A check has been added to ensure that the input is a non-empty vector.

TOB-AXIOM-17: Polynomial addition and subtraction assume polynomials have the same degree
Resolved. For both the addition and subtraction routines, `zip` has been replaced with `zip_eq` to ensure that the polynomials have the same degree.

TOB-AXIOM-18: FpChip::enforce_less_than_p incorrectly allows certain values above 2^t
Resolved. The `FpChip::enforce_less_than_p` function now requires a `ProperCrtUint` value, which is explicitly expected to guarantee that the `native` and `truncation` fields correspond to each other.

TOB-AXIOM-19: FpChip::assert_equal does not assert equality
Resolved. The `FpChip::assert_equal` function has been updated to properly check and assert equality.

TOB-AXIOM-20: Scalar rotation misbehaves on i32::MIN
Resolved. The rotation value is now cast to an `i64` before computing the negation.

TOB-AXIOM-21: Several functions assume that arguments are non-empty
Resolved. All of the functions enumerated in finding TOB-AXIOM-21 have been updated to ensure that the arguments are non-empty.

TOB-AXIOM-22: EVM verifier does not validate the deployment code
Resolved. The `compile_yul` function has been updated to assert that the binary is a non-empty vector, and the `split_by_ascii_whitespace` function has been updated to fix the issues.

TOB-AXIOM-23: Values from load_random_point are used without strict checks
Partially resolved. The `pippenger::multi_product` function has been commented out. The `pippenger::multi_exp_par` function has been updated to use the `ec_sub_strict` and `ec_sub_unequal` functions with `is_strict == true`. Both the `fixed_base_pippenger::multi_product` and the `fixed_base_pippenger::multi_exp` functions still contain non-strict-mode calls to `ec_add_unequal` and `ec_sub_unequal`.

TOB-AXIOM-24: query_cell_at_pos assumes that the column index is valid
Resolved. The `query_cell_at_pos` function has been updated to ensure that the column index equals the length of `self.columns`.

TOB-AXIOM-25: Unchecked uses of zip could bypass checks on parse_account_proof_phase0 and parse_storage_proof_phase0
Resolved. The use of `zip` has been replaced with `zip_eq` in both instances described in finding TOB-AXIOM-26.

TOB-AXIOM-26: The hex_prefix_encode and hex_prefix_encode_first functions assume that the is_odd parameter is a bit
Resolved. Documentation has been added to both functions to make it clear that `is_odd` is assumed to be a bit.

TOB-AXIOM-27: batch_invert_and_mul ignores zero elements and panics on empty arrays
Resolved. The function has been updated to ensure that the input array is non-empty, and the function now returns an error if it encounters non-invertible elements.

TOB-AXIOM-28: Proof caching occurs before proof validation

Resolved. The `gen_proof` function has been updated so that the proof is validated before being cached to a file, and the debug assertion has been replaced with a regular assertion.

TOB-AXIOM-29: Merkle root computation does not differentiate leaf data hashing and inner node hashing
Partially resolved. The implementation still does not use domain-separation, but a warning was added to inform users of this behavior.