MegaETH SALT
Security Review

Cantina Managed review by:

D Nevado, Lead Security Researcher

0xluk3, Associate Security Researcher
Jakubheba, Associate Security Researcher

January 12, 2026

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity level	Impact: High	Impact: Medium	Impact: Low
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings are a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

MegaETH is a high-performance blockchain network (currently in testnet) that is EVM-compatible and designed for ultra-fast block times and real-time applications.

From Sep 15th to Oct 7th the Cantina team conducted a review of salt on commit hash 0c2b4896. The team identified a total of **23** issues:

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	0	0	0
Medium Risk	4	1	3
Low Risk	10	5	5
Gas Optimizations	0	0	0
Informational	9	4	5
Total	**23**	**10**	**12**

3 Findings

3.1 Medium Risk

3.1.1 Non unique internal representation of `Element`

Severity: Medium Risk

Context: salt_committer.rs#L589

Summary: The elements of the Banderwagon subgroup can be internally represented in a non-unique way, yet this is accounted for only in some parts of the implementation.

Finding Description: This is not always true. Since double internal representation is allowed, the identity can be both `(0, 1)` and `(0, -1)`. We can get the first representation by calling `Element::zero()`, and the second with `Element::from_bytes(&[0u8;32])`. These lead to different encodings.

Impact Explanation: Multiple representations of points can lead to different serializations of the same point and faulty equality checks. These, in turn, can result in correctness and malleability issues for the protocols built on top.

Proof of Concept:

```
#[test]
fn batch_zero_element_to_commitments() {
    let zero = Element::from_bytes(&[0u8; 32]).unwrap();
    assert!(zero.is_zero());
    let hash_bytes_1 = Element::batch_to_commitments(&[zero])[0];
    let hash_bytes_2 = zero.to_bytes_uncompressed();
    dbg!(hash_bytes_1, hash_bytes_2);
    assert_eq!(hash_bytes_1, hash_bytes_2);

    let intended_bytes = Element::batch_to_commitments(&[Element::zero()])[0];
    assert_ne!(hash_bytes_1, intended_bytes);
}
```

Recommendation: Enforce a unique internal representation of the elements by controlling the entry points that allow the creation of the struct.

Cantina Managed: The issue persists. In particular, the proof of concepts shows how the zero element is still getting different representations. Marked as acknowledged.

3.1.2 Incorrect Banderwagon elements

Severity: Medium Risk

Context: element.rs#L9

Summary: `Element` can be created and hold incorrect Banderwagon elements.

Finding Description: The internal `EdwardsProjective` point should not be public. This allows the creation of `Element`s which do not satisfy:

- The curve equation.
- The subgroup check.

The upper layers (like the IPA commitment scheme), rely on the correctness of the points for their security guarantees.

Impact Explanation: Incorrect elements - elements that are not on the curve or in the appropriate subgroup - compromise the soundness of the vector commitment scheme and can also lead to malleability issues.

Likelihood Explanation: Elements are often created as a wrapper of an Edwards projective point, bypassing the crucial subgroup check.

Recommendation:

- Make the inner `EdwardsProjective` private.

- Only allow the creation of `Element` through designated methods that ensure the correctness of the element.

MegaETH: Fixed in PR 78.

Cantina Managed: Fix verified.

3.1.3 Errors are handled with panic affecting external integrations

Severity: Medium Risk

Context: types.rs#L286

Description: Libraries should never terminate execution through `panic!`, as doing so prevents callers from handling errors safely. In Rust, panics may abort the process, making it impossible for external software using the library to recover or report meaningful errors. When SALT is integrated as a reusable library within larger systems, callers must be able to receive structured error information (e.g., through `Result` or a `boolean` return) instead.

This is particularly problematic when user-supplied input can trigger the panic, as it enables external data to cause uncontrolled termination of the calling program and may lead to Denial of Service. Full list of potentially problematic functions if exposed outside of library (in progress).

- In `ipa_multipoint/src/crs.rs`, `from_bytes` expects at least one element to construct the CRS, otherwise panics.

```
#[allow(non_snake_case)]
pub fn from_bytes(bytes: &[[u8; 64]]) -> CRS {
    let (q_bytes, g_vec_bytes) = bytes
        .split_last()
        .expect("bytes vector should not be empty");
```

  In the same fine, `impl Index<usize>` for CRS will panic if the index i is out of bounds:

```
impl std::ops::Index<usize> for CRS {
    type Output = Element;

    fn index(&self, index: usize) -> &Self::Output {
        &self.G[index]
    }
}
```

- In `ipa_multipoint/src/lagrange_basis.rs`, `LagrangeBasis::evaluate_in_domain` panics on oob access:

```
pub fn evaluate_in_domain(&self, index: usize) -> Fr {
    self.values[index]
}
```

- In `salt/src/trie/trie.rs`, `StateRoot::rebuild` uses `.map_err(|_| unreachable!())` and additionally all public methods on `StateRoot` modify state, update, finalize, and update_fin use `expect()` in multiple places.
- In `salt/src/state/updates.rs` in `StateUpdates::add(&mut self, salt_key: SaltKey, old_value: Option<SaltValue>, new_value: Option<SaltValue>)`, panics in `assert_eq!(old_value, change.get().1, "Invalid state transition");`.
- In `salt/src/state/state.rs` functions: `EphemeralSaltState::shi_rehash` and `EphemeralSaltState::metadata` uses `expect()`.
- In `salt/src/trie/node_utils.rs` in `get_child_node(parent: &NodeId, child_idx: usize)` may lead to index out of bounds on `STARTING_NODE_ID`.
- In `salt/src/types.rs` several functions, for instance: `SaltValue::new` in `copy_from_slice`, `bucket_metadata_key` in `assert!(is_valid_data_bucket(bucket_id), ...)`, `bucket_id_from_metadata_key(key: SaltKey)` in `assert!(METADATA_KEYS_RANGE.contains(&key), ...)` and `pub fn is_subtree_node(node_id: NodeId)` can panic as well.

Recommendation: Consider changing the panic occurrences to `Error` types that can be returned to the caller.

MegaETH: Acknowledged. Won't fix all of them before the final report.

Cantina Managed: Acknowledged.

3.1.4 Missing consistency check between `b_vec` and `input_point`

Severity: Medium Risk

Context: ipa.rs#L253-L255

Summary: The `input_point` for IPA verification must be consistent with the input `b_vec`, which must hold the powers of the input point in Lagrange form.

Finding Description: There are no explicit checks that guarantee the consistency between `input_point` and `b_vec`. Moreover, `b_vec` never influences the transcript. So controlling this vector completely compromises the verification of the proof.

Moreover, this vector is directly fed into the `inner_product` function, which has "Unchecked length in inner product computation" that could be chained to this one.

Impact Explanation: A malicious or incoherent `b_vec` completely compromises verification.

Likelihood Explanation: It is unlikely that the issue gets exploited as the function is only called internally.

Recommendation: The verification function that is exposed by the library should not contain this input. It should receive only `input_point`. A possible solution is to make this function private and expose a wrapper that first computes the right `b_vec` and then calls the original function. It should also be clearly documented what each input of the function must be.

MegaETH: Acknowledged. Won't fix before the final report.

Cantina Managed: Acknowledged.

3.2 Low Risk

3.2.1 Unchecked length in inner product computation

Severity: Low Risk

Context: math_utils.rs#L5-L7

Summary: Different length inputs in `inner_product` lead to incorrect result.

Finding Description: The correctness of this function requires `a.len() == b.len()`. A short input can be passed in order to ignore some values of the other input.

Impact Explanation: A malicious short input in this function could be used to ignore or short-circuit values of the other input. This could be used to ignore challenges of the vector commitment scheme protocol and compromise the soundness of the proof.

Likelihood Explanation: This would need to be chained to other bugs in order to be exploited.

Recommendation:

- Assert the lengths of the inputs are equal.
- Reduce the visibility of the function.

MegaETH: Fixed in PR 79.

Cantina Managed: Fix verified.

3.2.2 Collisions in `map_to_scalar`

Severity: Low Risk

Context: element.rs#L132-L133

Summary: It is possible to find different, non-equivalent preimages of the same scalar. This map is not 2-to-1.

Finding Description: Different non-equivalent points can map to the same scalar field element. These points can be computed easily.

The `map_to_scalar_field` function maps a curve point into an element of the scalar field, but it is not 2-to-1.

It does so in 2 steps:

1. First, it maps the point to an element in the base field:

$$(x, y) \to x/y$$

   This function maps equivalent points in the Banderwagon subgroup to the same base field element.

2. Second, the base field element is mapped onto the scalar field by serializing it to bytes and then deserializing. Effectively, it is interpreting the base field as an integer in `[0, p-1]` and reducing it mod `r`. (Where `p` , `r` are the orders of the base and scalar fields respectively). Since `p` is several times larger than `r`, this map is not injective.

```
p = 0x73eda753299d7d483339d80809a1d80553bda402fffe5bfeffffffff00000001
r = 0x1cfb69d4ca675f520cce760202687600ff8f87007419047174fd06b52876e7e1`
```

Impact Explanation: Chained with other issues, this could be used to fake part of the path in the Verkle tree.

Likelihood Explanation: It is unlikely that the collision point can be used to build an exploit, since the underlying vector it commits too remains unknown.

Proof of Concept: `find_collision_point` takes an `Element` and returns (if possible) a different `Element` that maps to the same scalar as the original.

```
fn discriminant(k: Fq) -> Fq {
    let a = <BandersnatchConfig as TECurveConfig>::COEFF_A;
    let d = <BandersnatchConfig as TECurveConfig>::COEFF_D;
    let k_square = k.square();
    let four = Fq::from(4);

    (a * k_square + Fq::ONE).square() - four * d * k_square
  }

fn find_point_for_k(k: Fq) -> Option<Element> {
    // Curve parameters and constants.

    let a = <BandersnatchConfig as TECurveConfig>::COEFF_A;
    let d = <BandersnatchConfig as TECurveConfig>::COEFF_D;

    let k_square = k.square();

    let disc_sq = if let Some(x) = discriminant(k).sqrt() {
        x
    } else {
        // No solutions for k+ i*r
        return None;
    };

    let y_square =
        (a * k_square + Fq::ONE + disc_sq) * (Fq::from(2) * d * k_square).inverse().unwrap();

    let y = if let Some(y) = y_square.sqrt() {
        y
    } else {
        let y_square = (a * k_square + Fq::ONE - disc_sq)
            * (Fq::from(2) * d * k_square).inverse().unwrap();
        y_square.sqrt().expect("Should have found a valid y here")
    };

    let x = k * y;
    let p = EdwardsAffine::new_unchecked(x, y);
    return Some(Element(p.into()));
}

// Searches for a different element that serializes to the same scalar.
fn find_collision_point(e: Element) -> Option<Element> {
    let r_bytes = [
```

```
        0x1c, 0xfb, 0x69, 0xd4, 0xca, 0x67, 0x5f, 0x52, 0x0c, 0xce, 0x76, 0x02, 0x02, 0x68,
        0x76, 0x00, 0xff, 0x8f, 0x87, 0x00, 0x74, 0x19, 0x04, 0x71, 0x74, 0xfd, 0x06, 0xb5,
        0x28, 0x76, 0xe7, 0xe1,
    ];
    let r = Fq::from_be_bytes_mod_order(&r_bytes);

    //Input elem:
    println!("Input point:");
    println!("{}", e.0.into_affine());

    let mut k = e.map_to_field(); // x/y

    for i in 1..4 {
        k += r;
        if let Some(p) = find_point_for_k(k) {
            let p = p.0.into_affine();
            if p.is_on_curve() {
                println!("On curve!");
                if p.is_in_correct_subgroup_assuming_on_curve() {
                    println!("... and on subgroup!");
                    if p != e.0 && p != -e.0 {
                        println!("and different to input and its negation");
                        println!("Iteration {}. Found point:", i);
                        println!("{}", p);
                    }
                    return Some(Element(p.into()));
                }
            }
        }
    }

    None
}
```

Recommendation:

- Document properly the properties of this map.
- Ensure that all commitments in the protocol are opened at some point. This makes the collision points useless, because the underlying vector that commits to such points remains unknown.

Mitigation of this issue are discused in this blog post about the anatomy of a verkle proof.

MegaETH: Fixed in PR 78.

Cantina Managed: Fix verified.

3.2.3 Missing `Valid` implementation

Severity: Low Risk

Context: serialize.rs#L32

Summary: The implementation for `check()` in the trait `Valid` is missing, it just returns `Ok(())`.

Finding Description: There is no check of validity, `Ok` is returned for any element.

Impact Explanation: If an invalid element was passed as valid due to this error, the security guarantees of the vector commitment scheme would be comprised.

Likelihood Explanation: Since the trait is not currently being used, it is unlikely that the issue will have an impact.

Recommendation: One of the following:

- Drop this trait if it is unnecessary.
- Properly check the validity of the element by:
  1. Element is on the curve: It satisfies the curve equation.
  2. Subgroup check: In this case, a $2p$ size subgroup by checking $1 - ax^2$ is a quadratic residue.
  3. If a canonical representation is enforced: check the correct sign of y.

MegaETH: Fixed in PR 78.

Cantina Managed: Fix verified.

3.2.4 Faulty `ParitialEq` for `LagrangeBasis`

Severity: Low Risk

Context: lagrange_basis.rs#L8

Summary: Equal polynomials may appear as unequal.

Finding Description: Multiple representations are possible for the same polynomial. However, different representations will not appear as equal even if they represent the same polynomial.

Proof of Concept: These polynomials are the zero polynomial, and should be equal.

```
fn test_poly_eq() {
    let zero = Fr::from(0u128);
    let poly1 = LagrangeBasis::new(vec![zero, zero]);
    let poly2 = LagrangeBasis::new(vec![zero]);
    assert_eq!(poly1, poly2);
}
```

Recommendation: `PartialEq` should be implemented manually to account for the multiple possible representation of a polynomial. Alternatively, the `new` method could compute a canonical representation.

MegaETH: Acknowledged. Won't fix before the final report.

Cantina Managed: Acknowledged.

3.2.5 Faulty `Add` implementation for `LagrangeBasis`

Severity: Low Risk

Context: lagrange_basis.rs#L44

Summary: Addition of polynomials with different value length returns erroneous result.

Finding Description: It is necessary to check that `lhs.values` and `rhs.values` are the same length. If one of the input polynomial value vector is longer, some of the values of the other polynomial input are ignored resulting in an erroneous output.

Proof of Concept:

```
fn test_poly_add() {
    let zero = Fr::from(0u128);
    let one = Fr::from(1u128);
    let poly1 = LagrangeBasis::new(vec![zero, one]);
    let poly2 = LagrangeBasis::new(vec![zero]);

    let result1 = poly2.clone() + &poly1;
    let result2 = poly1 + &poly2;
    let expected = LagrangeBasis::new(vec![zero, one]);

    assert_eq!(result1, result2);
    assert_eq!(result1, expected);
}
```

Recommendation: Complete the shorter vector values with zeros up to the length of the largest. Then the implemented algorithm will return the correct result.

MegaETH: Acknowledged. We didn't fix it but added an explicit check in commit 52ea843e.

Cantina Managed: Acknowledged.

3.2.6 Generic error handling reduces interoperability

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Summary: Generic error types reduce library interoperability and make error handling difficult for external integrations.

Finding Description: Throughout the codebase, errors are returned with generic types (e.g., `ProofError::StateReadError`, `ProofError::InvalidLookupTable`) accompanied by debug-formatted string reasons. The error handling pattern follows such form:

```
.map_err(|e| ProofError::StateReadError {
    reason: format!("{e:?}"),
})?;
```

Occurrences:

- `salt/src/proof/prover.rs`: Generic `StateReadError` usage.
- `salt/src/proof/witness.rs`: Generic error handling in witness generation.
- `salt/src/proof/witness.rs`: `InvalidLookupTable` used for multiple distinct failure cases.
- `salt/src/state/state.rs`: Key mismatch and empty slot both treated as "not found".
- `salt/src/proof/subtrie.rs`: Storage read errors (for metadata, node entries, and commitments) are all collapsed into the same generic ProofError::StateReadError.
- `salt/src/proof/prover.rs:91`: `SerdeMultiPointProof` serialization logic converts I/O errors to generic serde errors via `to_string()`, hiding the original `std::io::Error` kind.
- `salt/src/state/state.rs`: `shi_rehash` fails silently by returning `Ok(())` in line 410 instead of an error when the provided `new_capacity` is insufficient, leaving the caller unaware that the operation was aborted.
- `salt/src/state/state.rs`: The `direct_find` helper function conflates multiple distinct outcomes (key not in the lookup table, key's value is empty, key mismatch from hash collision) into a single `Ok(None)` result.

Impact Explanation: Library consumers (external integrations, any software built on top) cannot distinguish between different failure modes like bucket metadata read errors, SHI search errors, key mismatches, or empty slots without fragile string parsing. This makes error handling difficult and reduces the reliability of integrating systems. Moreover, the `reason` format might differ among operating systems, which requires the integrators to handle errors differently depending on the OS.

Recommendation: A more optimal solution would be to throw errors related to the failure reason - which is to parse them here and return a corresponding error, for instance `BucketMetadataReadError`, `SHISearchError`, etc... Errors should be defined as self-explanatory, without additional `message` or `reason`.

MegaETH: Acknowledged. Won't fix all of them before the final report (fixed some).

Cantina Managed: Acknowledged.

3.2.7 The `Validate::Yes` path ignores validation

Severity: Low Risk

Context: serialize.rs#L65-L68

Description: The match handling of the validation flag routes both `Validate::Yes` and `Validate::No` to the same non-validating deserialization routine. As a result, callers that explicitly request validation receive none.

This can allow malformed, non-canonical, off-curve, or wrong-subgroup encodings to be accepted silently wherever validated decoding was expected, undermining soundness guarantees and any downstream assumptions that "validated bytes" were enforced.

Recommendation: Implement a truly validating branch for `Validate::Yes` that enforces canonical encoding, on-curve checks, and subgroup membership before constructing the element. Keep the non-validating branch only for trusted and internal use, and consider returning a `Result` from the validating path to propagate decoding errors cleanly.

MegaETH: Fixed in PR 78.

Cantina Managed: Fix verified.

3.2.8 Unchecked CRS point deserialization

Severity: Low Risk

Context: crs.rs#L80

Description: CRS loading uses an unchecked, uncompressed `Element::from_bytes_unchecked_uncompressed` point constructor for each entry even though the documentation says the byte array contains serialized elliptic curve points. Without canonical-encoding, on-curve, and subgroup checks, a tampered CRS can contain invalid points.

This can break binding, hiding properties and, in the worst case, enable proof forgeries or other algebraic traps if an attacker controls the CRS bytes.

Recommendation: Replace unchecked deserialization with a checked routine that verifies canonical encoding, on-curve status, and subgroup membership for every CRS point before acceptance. Refuse initialization on any failure and surface a descriptive error.

MegaETH: Fixed in PR 79.

Cantina Managed: Fix verified.

3.2.9 The `SaltValue::new` truncates lengths to u8 and lacks bounds checks

Severity: Low Risk

Context: types.rs#L286-L294

Description: The `new` constructor accepts arbitrary-length key and value slices, stores their lengths in single-byte fields, and then copies the original (`usize`) lengths into a fixed-size buffer. There are no checks that each length fits in a byte or that the total encoded size fits within the backing array.

This creates two problems:

1. Casting the lengths to `u8` can mis-encode any input longer than 255 bytes, producing headers that don't match the actual copied data.
2. If `2 + key_len + value_len` exceeds the buffer capacity, the slice copies will panic, yielding a reliable DoS when fed oversized inputs.

Downstream readers that trust the stored length bytes may then see inconsistent or malformed encodings.

Recommendation: Make the constructor fallible and validate inputs before writing. Return an explicit error on violation instead of panicking.

MegaETH: Acknowledged. Won't fix before the final report.

Cantina Managed: Acknowledged.

3.2.10 `SaltValue::key()` and `value()` trust embedded lengths without consistency checks

Severity: Low Risk

Context: types.rs#L299-L311

Description: The methods derive slice ranges from the first two bytes (key length and value length) and index directly into the buffer without verifying that `2 + key_len + value_len` fits within the actual buffer size. A maliciously crafted value can cause out-of-bounds slicing and a panic, resulting in a denial-of-service when processing untrusted witnesses.

Recommendation: Introduce a validated parsing routine that checks the buffer length before slicing and returns a `Result` on malformed inputs. Enforce that the total length equals the header-declared lengths and is within the maximum allowed size.

MegaETH: Acknowledged. Won't fix before the final report.

Cantina Managed: Acknowledged.

3.3 Informational

3.3.1 Non uniformly random CRS

Severity: Informational

Context: crs.rs#L191-L197

Summary: The method for deriving random points for the CRS does not follow a uniform distribution.

Finding Description: This method of generating a random coordinate is not uniform. It is deriving a random number from a uniform distribution in the range $[0, 2^{256})$ and then reducing it modulo `Fq::MODULUS`. The result is no longer uniform, in fact, it is significantly skewed:

```
2^256 = 2 * Fq::MODULUS + R
```

where `R` is `2^256 mod Fq::MODULUS`. Hence, the elements in `Fq` in the range `[0,R)` have a 50% higher probability of being drawn than the elements in the range `[R, Fq::MODULUS)`.

Impact Explanation: The impact is minimal. The CRS has not been generated in a completely uniformly random way but in practical terms there is no way to exploit this.

Recommendation: The usual approach to mitigate this issue is to generate at least 128 bits of extra randomness before performing the reduction. This way, the skewing is greatly reduced. Here is a detailed explanation and a reference implementation: https://github.com/zkcrypto/ff/blob/41fb01b8990909b8b1b44d4601556694c473bc16/src/lib.rs#L422-L448.

MegaETH: Acknowledged. Won't fix before the final report.

Cantina Managed: Acknowledged.

3.3.2 Non reflexive `Eq` in `Element`

Severity: Informational

Context: element.rs#L8

Summary: `Eq` derived for `Element` does not satisfy the reflexive property.

Finding Description: Deriving `Eq` requires the reflexive property. That means, that all `Element` is equal to itself. However, (due to the conditional checks at the beginning of `PartialEq` impl) this does not hold for the element `(0,0)`.This point is not on the curve so it should be impossible to create, so if `Element` guarantees the correctness of the point, this trait impl is fine.

Recommendation: Remove the implementation of `Eq`.

MegaETH: Fixed in PR 79.

Cantina Managed: Fix verified.

3.3.3 Doc error

Severity: Informational

Context: element.rs#L59-L60

Summary: Error in the documentation of equivalent points in Banderwagon.

Finding Description: This comment seems to be wrong:

> `(-x, y)` and `(x, y)` are different points in Banderwagon, so their conversion to bytes should return different byte arrays.

Recommendation: Change doc to:

> This is because if `(-x, -y)` is on the curve, then `(x, y)` is also on the curve. This method will return two different byte arrays for each of these.

MegaETH: Fixed in PR 78.

Cantina Managed: Fix verified.

3.3.4 Unconventional sign choice

Severity: Informational

Context: element.rs#L213

Summary: The designation of positive and negative elements in a prime field is arbitrary. Here, the convention chosen is the opposite of what is commonly used and is described in the original spec of BLS12-381 and Jubjub.

Finding Description: We are considering the lexicographical largest to be the positive, but this is the opposite of what the BLS specification suggests, which is: The values of `Fq` viewed as integers in $[0, p)$ are considered positive if the lie in the first half: $[0, \frac{p-1}{2}]$ and negative in the other half.

From page 11 in ZCash spec:

> "... means the positive square root of a in F_q, i.e. in the range $\{0..\frac{q-1}{2}\}$. ".

The current implementation satisfies:

```
is_positive( - Fq::ONE ) == true
```

which can be misleading.

Impact Explanation: Minimal. Just counter-intuitive behavior.

Likelihood Explanation: Minimal. It may lead to conflicting implementations of serialization, but it is unlikely.

Recommendation: Document clearly what is meant by positive and how this affects the chosen representative of Banderwagon elemens and their serialization.

MegaETH: Acknowledged. Won't fix.

Cantina Managed: Acknowledged.

3.3.5 Non uniform transcript challenges

Severity: Informational

Context: transcript.rs#L47-L53

Summary: The modular reduction used to obtain scalar challenges from the transcript breaks the uniformity of the distribution.

Finding Description: It would be good to get more bits of entropy to generate the scalar in order to reduce the bias and have the challenge distribution be closer to uniform.

Recommendation: Generate a random number between $[0, 256 + 128)$ and reduce it modulo r to obtain the challenge scalar.

MegaETH: Acknowledged. Won't fix before the final report.

Cantina Managed: Acknowledged.

3.3.6 Constant description is inconsistent with implementation

Severity: Informational

Context: trie.rs#L924

Description: In `salt/src/trie/trie.rs:924`, code comment states empty entries are hashed to `0`, however, the actual implementation in constants.rs defines `pub const EMPTY_SLOT_HASH: [u8; 32] = [1u8; 32]` ; which means that the array is filled with 1's.

Recommendation: While no security impact was found, it is recommended to adjust the comment to the actual state of the code.

MegaETH: Fixed in PR 78.

Cantina Managed: Fix verified.

3.3.7 Comparison can be simplified

Severity: Informational

Context: prover.rs#L39

Description: The `PartialEq` implementation for `SerdeCommitment` uses unchecked deserialization, which is safe but unnecessarily complex. In `salt/src/proof/prover.rs:39`, the `eq` implementation directly compares raw bytes with `from_bytes_unchecked_uncompressed`. It differs from the deserialization path (e.g., in deserialize), which first calls `Element::from_bytes for validation`. While this is safe in context, it introduces unnecessary complexity to the comparison logic.

Recommendation: Consider changing `SerdeCommitment` to wrap an `Element` directly.

MegaETH: Fixed in PR 79.

Cantina Managed: Fix verified.

3.3.8 Unbounded table index in `add_deltas`

Severity: Informational

Context: salt_committer.rs#L223-L236

Summary: The `add_deltas` method uses the provided index to index into internal tables without bounds checks. If caller input were untrusted, an out-of-range index could cause a panic or memory access error. In this codebase, the function is currently used only in unit tests and need to be moved under a test module, so it does not present a production risk.

Recommendation: No production change required if the function remains test-only. If it is ever re-introduced into non-test code, add explicit bounds checks (or safe indexing returning a `Result`) and treat indices as untrusted.

MegaETH: Acknowledged. Moved into test module just to be clear.

Cantina Managed: Acknowledged.

3.3.9 Unconventional use of CRS point `Q`

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Summary: The `Q` point in the CRS is not being used for its intended purpose.

Finding Description: The CRS provides a blinder point `Q` that allows to add zk to when using IPA as a PCS. However, `Q` is being used for a different purpose. It is being used as a generator for the challenge point of the IPA protocol. Since the dlog relation between `Q` and the `G_i` points in the CRS is unknown, this is not a problem for security.

Recommendation: Since ZK is not needed, the doc for this point can be updated to reflect its actual purpose. This is a good reference implementation for the parameters needed in IPA: https://github.com/zcash/halo2/blob/2308caf68c48c02468b66cfc452dad54e355e32f/halo2_proofs/src/poly/commitment.rs#L26-L33.

MegaETH: Acknowledged. Won't fix before the final report.

Cantina Managed: Acknowledged.