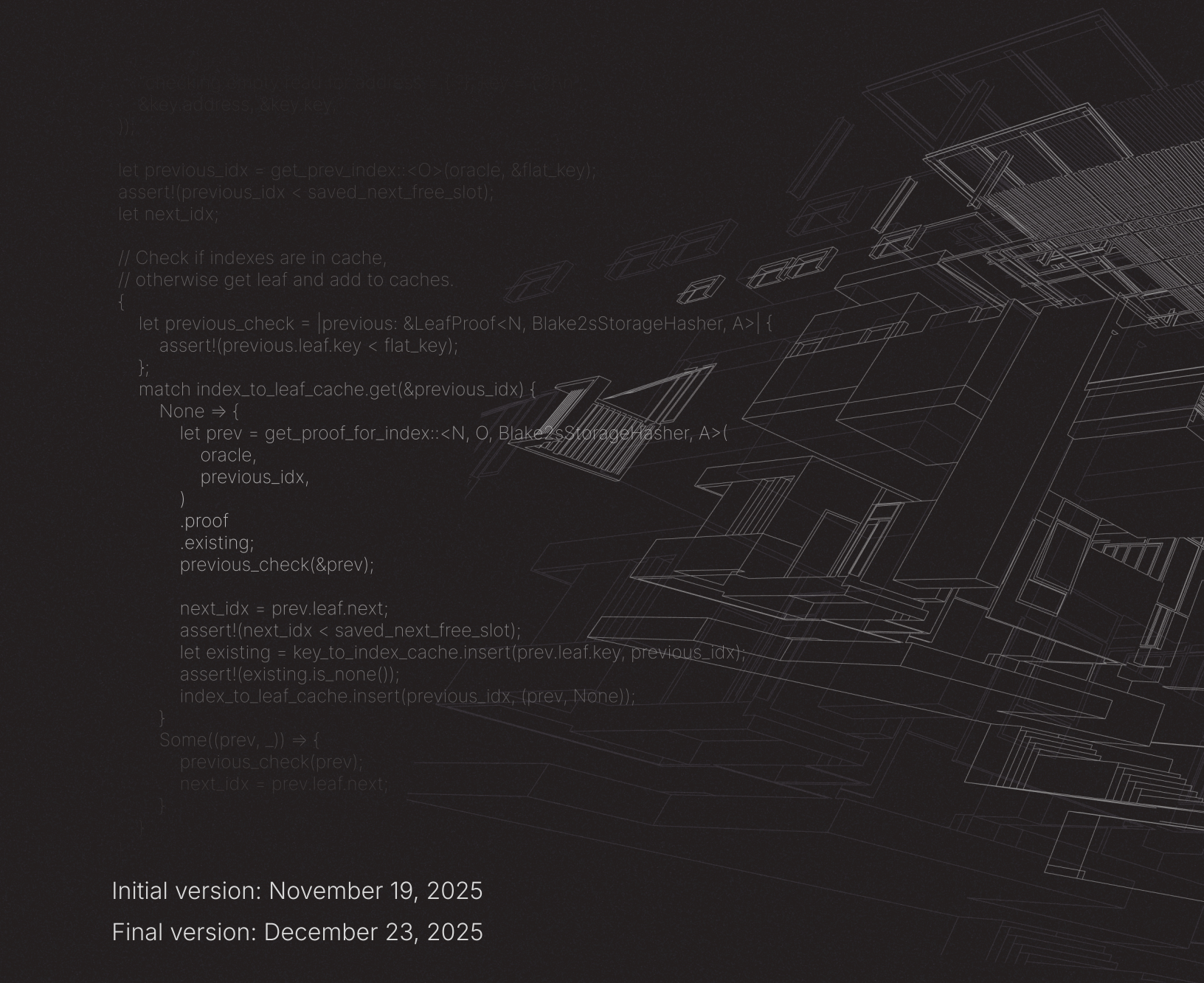
TARAN SPACE

ZKsync

SECURITY AUDIT REPORT

of project **ZKsync OS - Crypto** developed by **Matter Labs**



Initial version: November 19, 2025

Final version: December 23, 2025

TABLE OF CONTENTS

LICENSE

DISCLAIMER

The information contained within this audit report is made available on an "AS IS" basis, without any representations or warranties, express or implied. Neither the author(s) nor their employing entity shall be held liable for any losses or damages, direct or indirect, arising from or in any way connected to the use of this report. This document has been developed solely for the benefit and internal use of the specified client and does not establish any form of legal relationship with any third parties. The author(s) and their employer expressly disavow any liability or duty to any third parties, refusing to guarantee the report's accuracy or completeness. It is important to clarify that this report does not introduce any further obligations upon the company, including, but not limited to, warranties or liabilities. The copyright over this audit report is retained by the author(s).

This audit has been performed by:

Taran Space Co., Ltd. (ENG)
주식회사 타란스페이스 (KOR)

https://taran.space
info@taran.space

ABOUT THE AUDITOR COMPANY

Taran Space is a security research team offering reviews and other security-related services to Web3 projects, with the ultimate goal of fostering a stronger ecosystem. We conduct reviews of Layer 1 and Layer 2 protocols, cross-chain mechanisms, zero-knowledge circuits, web applications, mobile wallets, off-chain indexers, and more.

Importantly, Taran Space is not backed by any venture capital (VC) funds.

Discover more about us at our website **taran.space**.

Follow **@taran_space on X/Twitter**

Follow **@taran.space on BlueSky**

See other reports in our GitHub repository:
github.com/taran-space/audit-reports

If you're interested in exploring partnership opportunities with Taran.Space, please reach out to us at **hello@taran.space**.

UNDERSTANDING AUDIT LIMITS

While we diligently strive to identify as many issues as possible, by thoroughly examining every line of code, it is important to keep in mind that an audit may not uncover all potential vulnerabilities. No audit, including ours, can offer an absolute guarantee against the possibility of a hack. This inherent limitation stems from several factors:

- An audit represents a snapshot in time, capturing the state of the client's project at a specific commit. Given the dynamic nature of almost all software systems, which evolve and change over time, any modifications to the code post-audit necessitate a new security review to ensure continued safety.
- Even relatively small codebases can present a vast array of potential scenarios and code execution paths, each with its own complexities. This diversity means that there could always be parts of the attack surface that remain unexplored, no matter how comprehensive the audit.
- In the real world, the attack surface extends beyond the code itself to include deployment scripts, the supply chain, private key management systems, and potentially other components. To provide a more complete security assessment, such components should explicitly be included in the scope.
- Human factors also play a critical role in system security. Operators may alter deployment scripts, craft custom commands and governance proposals, or inadvertently compromise security through errors or oversights. While we endeavor to report any issues related to centralization, these may not always be flagged as of the highest severity. It is crucial for human operators to carefully review any scripts or governance proposals before executing them.

We have exerted our utmost effort to uncover all vulnerabilities within the system. However, our findings should not be considered an exhaustive list of all potential issues. The information provided in this document is for informational purposes only and should not be construed as investment or legal advice. The authors of this document cannot be held responsible for any decisions made based on its content.

Synopsis

In October 2025, Matter Labs engaged Taran Space to conduct an additional security assessment of their new product, ZKsync OS, with a focus on the cryptographic modules. The audit was conducted over a single 4-week phase.

Following the assessment, Matter Labs addressed the most severe security issues, and the implemented fixes were verified in December 2025.

About ZKsync OS

ZKsync OS, developed by Matter Labs, is a generalized RISC-V–based state transition function used within the ZKsync protocol. Further details are available in our previous report: https://github.com/taran-space/audit-reports/blob/main/2025-08-ZKsync-OS.pdf

This audit focused on ECC and field arithmetic in ZKsync OS: Secp256k1 and Secp256r1 implementations, BN254 and BLS12-381 fields and pairings, big-integer delegation on RISC-V, normalization and magnitude invariants, and point/field operations used in proof verification.

Audit Team

The audit of ZKsync OS was conducted by a team of two full-time auditors:

- Lead Auditor: Kirill Taran
- Auditor: Ilia Imeteo

Fuzz-testing assistance was provided by:

- Auditor: Tarek Elsayed

Scope and Timeline

Repository

The codebase is located in the repository by the following link:

- https://github.com/matter-labs/zksync-os

Modules in Scope

The scope is limited to specific modules:

- `callable_oracles`
- `crypto`

Timeline

Duration: 4 weeks

Dates: October 20, 2025 - November 19, 2025

Commit hash:

- `f32e78b857f699cdfa30c3e46a35ccb177f67a89`

Findings Classification

Severity Rating

The severity rating is assigned to each issue after evaluating two factors: "likelihood" and "impact".

Likelihood

Likelihood reflects the ease of exploitation, indicating how readily attackers could exploit a finding. This assessment considers the required level of access, the availability of exploitation information, the necessity of social engineering, the presence of race conditions or brute-forcing opportunities, and any additional barriers to exploitation.

This factor is categorized into one of three levels:

- **High** Exploitation is almost certain to occur, straightforward to execute, or challenging but highly incentivized. The issue can be exploited by virtually anyone under almost any condition. Attackers can exploit the finding unilaterally without needing special permissions or encountering significant obstacles.
- **Medium** Exploitation of the issue requires non-trivial preconditions. Once these preconditions are met, the issue is either easy to exploit or well incentivized. Attackers may need to leverage a third party, gain access to non-public information, exploit a race condition, or overcome moderate challenges to exploit the finding.
- **Low** Exploitation requires stringent preconditions, possibly requiring the alignment of several unlikely factors or a preceding attack. It might involve implausible social engineering, exploiting a challenging race condition, or guessing difficult-to-predict data, making it unlikely. There is little or no incentive to exploit the issue. Centralization issues, exploitable only by operators, on-chain governance, or development teams, fall into this category as well.

Impact

Impact considers the consequences of successful exploitation on the target system. It accounts for potential loss of confidentiality, integrity, and availability, as well as potential reputational damage.

This factor is classified into one of three levels:

- **High** Entails the loss of a significant portion of assets within the protocol, or causes substantial harm to the majority of users. Attackers can access or modify all data in a system, execute arbitrary code, escalate their privileges to superuser level, or disrupt the system's ability to serve its users.

- **Medium** Leads to global losses of assets in moderate amounts or affects only a subset of users, which is still deemed unacceptable. Attackers can access or modify some unauthorized data, affect availability or restrict access to the system, or obtain significant internal technical insights.
- **Low** Losses are inconvenient but manageable. This applies to scenarios like easily remediable griefing attacks or gas inefficiencies. Attackers may access limited amounts of unauthorized information or marginally degrade system performance. This category also includes code quality concerns and non-exploitable issues that may still negatively impact public perception of security.

Final Severity

Upon assessing the issue's likelihood and impact, its severity is determined using the table below:

	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	QA

In exceptional cases, the total severity may be elevated to highlight the significance of the issue. Whenever this occurs, the rationale for applying an increased severity level is detailed in the report.

Issue Statuses

The status of an issue can be one of the following:

- Notified indicates that the client has been informed of the issue but has either not addressed it yet or has acknowledged the behavior without planning to remediate it soon. This inaction may stem from the client not viewing the behavior as problematic, lacking a feasible solution, or intending to address it at a later date.
- Addressed is used when the client has made an effort to address the issue with partial success. In the case of complex issues, the provided fix may only resolve one of several points described.
- Fixed means that the client has implemented a solution that completely resolves the described issue.

Summary of Issues

Title	Severity	Status
1. Incorrect identity predicates for affine and Jacobian points	Critical	Fixed
2. DoS and corrupted state in BN254 and BLS12-381 pairings checks	Critical	Fixed
3. Insufficient magnitude update in scalar multiplication	High	Fixed
4. Failed assertion in the Fused Multiply-Add operation	Medium	Fixed
5. Unchecked field-arithmetic invariants	Medium	Addressed
6. Magnitude limit 2047 cannot detect arithmetic issues in 5×52 backend	Low	Fixed
7. Incorrect validation of wNAF table digits	Low	Fixed
8. Miscellaneous validation issues	Low	Addressed
9. Concurrent usage is not supported	QA	Notified
10. Variable-time scalar operations should not be used with secret data	QA	Addressed
11. Normalization check is not strong enough	QA	Notified
12. Unit tests issues	QA	Addressed
14. Excessive magnitude increment	QA	Notified
15. Magic numbers decrease maintainability	QA	Addressed
16. Code duplicates	QA	Notified
17. Redundant `else` branch	QA	Notified

Detailed List of Issues

1 Incorrect identity predicates for affine and Jacobian points

Severity: Critical Impact: High Likelihood: High Status: Fixed

In the `secp256r1` module, incorrect predicates are used to determine whether a point represents the identity (the point at infinity):

src/secp256r1/points/jacobian.rs
```
282   pub(crate) const fn is_infinity_const(&self) -> bool {
283       self.z.is_zero() || (self.x.is_zero() || self.y.is_zero())
284   }
```

src/secp256r1/points/affine.rs
```
21    pub(crate) fn is_infinity(&self) -> bool {
22        self.infinity || (self.x.is_zero() || self.y.is_zero())
23    }
```

In Jacobian coordinates, a point represents the identity if and only if `z` is zero. In affine coordinates, the identity is typically represented only using an explicit flag, not by coordinate values.

While interpreting the `(0, 0)` as the identity could be intended as a simplification of the implementation, the aforementioned functions also consider points `(0, y)` and `(x, 0)` as the identity. This is logically incorrect: valid curve points may legitimately have `x` or `y` equal to zero, and must not be treated as the identity.

This flaw breaks fundamental elliptic-curve invariants and has critical impacts:

- Valid signatures may be rejected
- Invalid signatures may be accepted

This behavior diverges from standard `secp256r1` implementations and can lead to inconsistent public keys, hashes, and verification results. Arithmetic operations may return incorrect results, spuriously produce the identity, or terminate early: addition, doubling and, overall, scalar multiplication.

The following functions rely directly or indirectly on the incorrect identity predicates:

- Function `odd_multiples`, defined in `context.rs:29-41`
- Function `double`, defined in `jacobian.rs:297-298`
- Function `add`, defined in `jacobian.rs:320-324`
- Function `reject_identity()`, defined in `affine.rs:54-59`
- Function `double_assign`, defined in `jacobian.rs:36-81`
- Function `add_ge_assign`, defined in `jacobian.rs:159-220`

Recommendations

1. As an immediate mitigation, replace the second `||` operator with `&&`, so that points with exactly one zero coordinate are no longer misclassified as the identity. This reduces incorrect behavior but does not fully restore the correct mathematical definition of the point at infinity.
2. A mathematically correct approach is to define the identity exclusively via the coordinate system invariant: `z` is zero for Jacobian points, or the explicit `infinity` flag for affine points, and never determine identity from `(x, y)` coordinate values.
3. Centralize identity checks into a single helper function shared by affine and Jacobian representations to prevent future inconsistencies.

This issue has been resolved completely.

2 DoS and corrupted state in BN254 and BLS12-381 pairings checks

Severity: Critical Impact: High Likelihood: High Status: Fixed

The issue in the `bn254` module

The function **`bn254_pairing_check_inner`** parses the buffer **`src`** into G1 and G2 affine points and constructs **`(G1Affine, G2Affine)`** pairs. The buffer **`src`** is supplied by the caller function **`execute`** and may contain arbitrary data defined during a smart contract execution. The **`(G1Affine, G2Affine)`** pairs are then passed into the function **`Bn254::multi_pairing`** on line **`154`**:

```
basic_system/src/system_functions/bn254_pairing_check.rs
66   fn bn254_pairing_check_inner<A: Allocator>(
67       num_pairs: usize,
68       src: &[u8],
69       allocator: A,
70   ) -> Result<bool, ()> {
```

```
152   let g1_iter = pairs.iter().map(|(g1, _)| g1);
153   let g2_iter = pairs.iter().map(|(_, g2)| g2);
154   let result = Bn254::multi_pairing(g1_iter, g2_iter);
```

The function **`Bn254::multi_pairing`** includes a call to the **`Bn254::multi_miller_loop`** function, which processes each *(G1,G2)* pair by converting the *G1* input into **`G1Prepared`** and the **`G2Affine`** input into **`G2Prepared`**. After the conversions, both results are checked for representing the point-at-infinity, using the **`is_zero`** function. If either point is the point-at-infinity, the pair is skipped and the loop continues with the next pair:

```
crypto/src/bn254/curves/pairing_impl.rs
109   let q: Self::G2Prepared = q.into();
110   if q.is_zero() {
111       continue;
112   }
```

The conversion **`into()`** for **`G2Affine`** type is implemented via **`G2PreparedNoAlloc::from`**, which handles the point-at-infinity case as follows:

```
impl From<G2Affine> for G2PreparedNoAlloc
217   if q.infinity {
218       #[allow(invalid_value)]
219       G2PreparedNoAlloc {
220           ell_coeffs: unsafe { core::mem::MaybeUninit::uninit().assume_init() }, //
unused/filtered above
221           infinity: true,
222       }
223   } else {
```

Both **`q.is_zero()`** and **`q.infinity`** implement the check for point-at-infinity:

```
357   impl G2PreparedNoAlloc {
358       pub fn is_zero(&self) -> bool {
359           self.infinity
360       }
361   }
```

This means that both lines **`110`** and **`217`** check for the same condition, so there is a scenario when the block on lines **`218-222`** is executed. The line **`220`** contains an **`unsafe`** expression, accompanied by the comment **`// unused/filtered above`**. Apparently, the idea here is that the **`unsafe`** section would never actually execute if the field **`ell_coeffs`** is not accessed. However, struct fields are eagerly initialized in Rust, so all fields of a struct literal are evaluated before the value exists, and the **`unsafe`** section is executed every time the control flow reaches lines **`218-222`**.

In other words, the **`unsafe`** section on line **`220`** is always entered before the filtering on line **`110`** occurs.

This **`unsafe`** section immediately triggers an Undefined Behavior, even if the field **`ell_coeffs`** is never accessed later, because the expression **`MaybeUninit::uninit().assume_init()`** creates an uninitialized array and treats it as initialized. This is confirmed to be an invalid use by the documentation of the **`assume_init`** function:

```
/// # Safety
///
/// It is up to the caller to guarantee that the `MaybeUninit<T>` really is in an initialized
/// state. Calling this when the content is not yet fully initialized causes immediate undefined
/// behavior. The [type-level documentation][inv] contains more information about
/// this initialization invariant.
```

Similar issue in the `bls12_381` module

The same issue is present in the BLS12-381 implementation, due to code duplication.

crypto/src/bls12_381/curves/pairing_impl.rs

```
fn multi_miller_loop
112   let q: Self::G2Prepared = q.into();
113   if q.is_zero() {
114       continue;
115   }
```

```
impl From<G2Affine> for G2PreparedNoAlloc {
217   if q.infinity {
218       #[allow(invalid_value)]
219       G2PreparedNoAlloc {
220           ell_coeffs: unsafe { core::mem::MaybeUninit::uninit().assume_init() }, //
unused/filtered above
221           infinity: true,
222       }
223   } else {
```

Severity

The impact of Undefined Behavior in a cryptographic setting is considered High, because it invalidates all guarantees about correctness, soundness, reproducibility, and security. Undefined Behavior can affect any code surrounding the line triggering it, since it permits the compiler to apply otherwise invalid optimizations.

Practically, UB may allow the attacker to:

- bypass cryptographic verification completely
- corrupt other transactions in any blocks within the same execution context

Since input points are derived from attacker-controlled smart contract calldata, triggering the UB is trivial. Exploitability also depends on compiler behavior but we classify the likelihood of this issue as High.

Even if the corrupted state is eventually rejected during verification, the issue still equips attackers with an extremely cheap DoS vector.

Recommendations

Apply these recommendations to both BN254 and BLS12-381 implementations:

1. Reject invalid points, including the point-at-infinity, early — before invoking any conversions on them.
2. Construct a concrete, fully-initialized dummy `EllCoeff` (zeros) in the array initializer.

This issue has been resolved completely.

3 Insufficient magnitude update in scalar multiplication

Severity: High Impact: High Likelihood: Medium Status: Fixed

In the `secp256k1` module, several incorrect magnitude updates have been identified, specifically in the implementation of scalar multiplication.

The function `mul_int_in_place`, which takes a field element `self` and a scalar `rhs`, increases the current magnitude by `rhs`. This is incorrect: every limb of the field element is multiplied by `rhs`, so if a limb is already inflated by a factor of `k`, the resulting inflation factor is `k * rhs`, not `k + rhs`.

src/secp256k1/field/field_impl.rs

```
pub(super) fn mul_int_in_place(&mut self, rhs: u32)
79    self.magnitude += rhs;
80    debug_assert!(self.magnitude <= Self::max_magnitude());
81
82    self.value.mul_int_in_place(rhs);
83    self.normalized = false;
```

The same incorrect update is present in `mul_int`:

```
pub(super) const fn mul_int(&self, rhs: u32)
159   let new_magnitude = self.magnitude + rhs;
160   debug_assert!(new_magnitude <= Self::max_magnitude());
161
162   let value = self.value.mul_int(rhs);
163   Self::new(value, new_magnitude)
```

Both `mul_int_in_place` and `mul_int` underestimate the resulting magnitude of the field element. At a minimum, this invalidates assumptions relied upon by routines such as `negate_in_place`.

In this codebase, the magnitude invariants are ensured only by debug assertions, and no runtime validations exist in release mode. As a consequence, silent field-arithmetic failures may occur. In the context of signature verification, this may lead to:

- invalid signatures being accepted,
- valid signatures being rejected.

If the same code were used in a decentralized environment (e.g. an L1 network), an additional impact would be the risk of consensus faults.

Recommendation

Correct the magnitude update so that the resulting magnitude is computed as the product of the previous magnitude and the scalar, rather than their sum.

This issue has been resolved completely.

4 Failed assertion in the Fused Multiply-Add operation

Severity: Medium Impact: High Likelihood: Low Status: Fixed

During fuzzing test runs, an assertion failed:

```
thread '<unnamed>' panicked at /zksync-
os/basic_system/src/system_functions/modexp/delegation/bigint.rs:686:25:
assertion `left == right` failed
  left: 1
 right: 0
```

See the *Appendix* section for details on how fuzzing was performed.

The assertion is located in the function `fma`, which implements a Fused Multiply-Add (FMA) operation for the `BigintRepr` structure.

The function `bigint_op_delegation_raw`, defined in the in-scope crate `crypto`, returns a non-zero value to signal a limb overflow. The assertion assumes that no overflow can occur at this stage of the algorithm:

```
basic_system/src/system_functions/modexp/delegation/bigint.rs
unsafe fn fma
681   let of = bigint_op_delegation_raw(
682       dst_scratch_capacity[dst_digit].as_mut_ptr().cast(),
683       carry_scratch.cast(),
684       BigIntOps::Add,
685   );
686   assert_eq!(of, 0);
```

However, this assumption is incorrect. During accumulation, under certain operand combinations, a carry produced by earlier partial products can overflow the destination digit, requiring propagation into higher digits.

As a consequence, this creates a limited potential for panic-based DoS during modular exponentiation. Because this is a non-debug assertion, the failure is a deterministic panic in release builds rather than silent truncation. The function `fma` is callable from functions `mul_step`, `square_step`, and `reduce_initially`. These functions are invoked by the `modpow` function, which is reachable from the top-level `execute` function when processing the `modexp` system function. This system function can be used by both honest and malicious transactions. If the assertion is triggered by an honest transaction, the failure will appear as a DoS to the user.

At the same time, the likelihood of exploitation is low. The failing input is not trivially constructible.

In the function `fma`, carry propagation on non-zero limb overflow has been implemented.

5 Unchecked field-arithmetic invariants

Severity: Medium Impact: High Likelihood: Low Status: Addressed

Regular CI test runs execute the test suite only in Release mode, where debug assertions and integer overflow checks are disabled. The codebase provides no runtime validation of field-arithmetic invariants in Release builds. Correctness of the field operations depends entirely on debug-mode assertions and

overflow checks, which are never exercised by CI. If these invariants are violated, the failures remain silent in Release mode, allowing field-arithmetic errors to persist undetected.

Consider the file **`.github/workflows/ci.yml`**:

- The only **`cargo test`** commands without **`--release`** flag used in this file are located on lines **`71`** and **`155`**, where the scope of their action is limited to modules **`transactions`** and **`binary_checker`** correspondingly.
- All the other invocations of **`cargo test`** run in Release profiles: lines **`43`**, **`45`**, **`178`**, **`201`** and **`202`**.

As a consequence, overflows and debug assertions during test suite execution are not reported.

When running the tests specifically on the **`crypto`** crate, with overflow checks and debug assertions enabled, multiple sporadious issues have been discovered. These issues do not reproduce consistently due to randomized nature of property testing involved.

Proof of concept

After running **`./dump_bin.sh`** as usually, enable the extra checks using the **`RUSTFLAGS`** and build the project including the test code of the **`crypto`** crate:

```
export RUSTFLAGS="-C overflow-checks=on -C debug-assertions=on"
cargo build --release
cargo test --release -p crypto
```

Then run the property tests multiple times, using a loop:

```
$ for i in `seq 1 1000`
do
    cargo test --release -p crypto 2>&1 | grep FAILED | tee -a errors.txt
done
```

During manual exploration, it is practical to limit scope of each pass to only one submodule or one test case, for easier interpretation of results. Additionally, parallelization should be enabled in order to save total computation time:

```
seq 1 1000 \
| xargs -n1 -P48 sh -c 'cargo test --release -p crypto secp256k1::field::field_10x26 2>&1 | grep
FAILED || true' _ \
| tee -a errors.txt
```

Examples of unreported issues

Although it is recommended to ensure overflow checks and debug assertions in all test cases, actual violations have been discovered only in the `secp256k1::field` module.

Violated assertions

These test cases have crashed on a `debug_assert!` statement:

- `secp256k1::field::field_10x26`: `test_add`, `test_invert_var`, `test_mul`, `to_bytes_round`, `to_signed30_round`
- `secp256k1::field::field_5x52`: `test_add`, `test_invert`, `test_mul`, `to_bytes_round`, `to_signed62_round`, `to_storage_round`
- `secp256k1::field::tests`: `storage_round_trip`, `test_add`, `test_add_const`, `test_invert`, `test_invert_const`, `test_mul`, `test_mul_const`, `test_square`, `test_square_const`, `to_bytes_round_trip`
- `secp256k1::points::jacobian`: `test_add_zinv`

For instance, one of violated debug assertions is located in the `normalize_in_place` function:

```
src/secp256k1/field/field_10x26.rs
pub(super) const fn normalize_in_place(&mut self) {
589   self.0[8] &= 0x3FFFFFF;
590   m &= self.0[8];
591
592   /* ... except for a possible carry at bit 22 of self.0[9] (i.e. bit 256 of the field element)
*/
593   debug_assert!(self.0[9] >> 23 == 0);
594
595   // At most a single final reduction is needed; check if the value is >= the field
characteristic
596   x = (self.0[9] >> 22)
597       | ((self.0[9] == 0x03FFFFF)
598           & (m == 0x3FFFFFF)
599           & ((self.0[1] + 0x40 + ((self.0[0] + 0x3D1) >> 26)) > 0x3FFFFFF))
600           as u32;
```

Missed overflows

These test cases have crashed with overflow error:

- `secp256k1::field::field_10x26`: `test_add`, `test_invert_var`, `test_mul`, `to_bytes_round`, `to_signed30_round`

- `secp256k1::field::field_5x52`: `test_add`
- `secp256k1::field::tests`: `test_add_const`
- `secp256k1::points::affine`: `jacobian_round_trip`
- `secp256k1::points::jacobian`: `test_add_zinv`

For instance, the overflows error in the `field_10x26::test_mul` test case is thrown from the `normalize_in_place` function:

```
src/secp256k1/field/field_10x26.rs
pub(super) const fn normalize_in_place(&mut self) {
559   // Reduce self.0[9] at the start so there will be at most a single carry from the first pass
560   let mut x = self.0[9] >> 22;
561   self.0[9] &= 0x03FFFFF;
562
563   // The first pass ensures the magnitude is 1, ...
564   self.0[0] += x * 0x3D1;
565   self.0[1] += x << 6;
566   self.0[1] += self.0[0] >> 26;
567   self.0[0] &= 0x3FFFFFF;
```

It is possible that some of the overflows are caused by broken invariants, indicated by the violated assertions.

Recommendation

Refactor CI workflows and run all tests with all possible checks since running the test suite on local developer's machine is not straightforward and time consuming. In security context, this means that the test suite is never actually executed to the full extent.

Additionally, run the test suite multiple times. Property-based tests do not guarantee full coverage in a single pass, and repeated execution significantly increases the likelihood of catching regressions.

This is a meta-issue. CI workflows have been updated to run tests with debug assertions and overflow checks enabled. The failures described in this issue were caused by incorrect test-input generation, which has now been fixed as well. The issue was classified with "Medium" severity only because debug assertions are the sole enforcement mechanism for safety invariants; runtime checks are avoided for performance reasons.

6 Magnitude limit 2047 cannot detect arithmetic issues in 5×52 backend

Severity: Low Impact: Medium Likelihood: Low Status: Fixed

The magnitude limit for the `5x52` storage is defined as `2047`:

```
crypto/src/secp256k1/field/field_5x52.rs
89    pub(super) const fn max_magnitude() -> u32 {
90        2047u32
91    }
```

Although `2047` is the mathematical upper bound for keeping all limbs within 63 bits, it is not compatible with the actual algorithmic invariants in the `mul_in_place` implementation.

The `mul_in_place` implementation contains tight limb-width debug assertions:

```
crypto/src/secp256k1/field/field_5x52.rs
pub(super) const fn mul_in_place
162   debug_assert!(a0 >> 56 == 0);
163   debug_assert!(a1 >> 56 == 0);
164   debug_assert!(a2 >> 56 == 0);
165   debug_assert!(a3 >> 56 == 0);
166   debug_assert!(a4 >> 52 == 0);
```

These constraints imply:

- Limbs 0–3 may temporarily grow from 52 bits up to 56 bits.
- Limb 4 may expand only to 52 bits.
- No limb is ever allowed to approach the theoretical 63-bit capacity that would justify a magnitude limit of `2047`.

Thus, a magnitude of `2047` can never occur during legal operation, and using `2047` as `max_magnitude()` renders magnitude-based invariants validation ineffective. Internal arithmetic faults may pass unnoticed.

Estimation based on `mul_in_place` invariants

A normalized limb less than **`2^52`** corresponds to the magnitude of 1. The largest permitted limb before entering **`mul_in_place`** is less than **`2^56`**. Thus the true maximum magnitude allowed by the **`mul_in_place`** function is:

```
(2^56) / (2^52) = 2^4 = 16
```

Any magnitude above **`16`** cannot be produced by correct arithmetic and cannot be safely consumed by **`mul_in_place`**.

Recommendation

Set the magnitude limit to **`16`**.

This issue has been resolved completely.

7 Incorrect validation of wNAF table digits

Severity: Low Impact: Medium Likelihood: Low Status: Fixed

In **`secp256k1`**, the function **`table_verify`** is used in debug assertions to validate wNAF digits:

crypto/src/secp256k1/recover.rs

```
328   fn table_get_ge(pre: &[Affine], n: i32, w: usize) -> Affine {
329       debug_assert!(table_verify(n, w));
330
331       if n > 0 {
332           pre[(n - 1) as usize / 2]
333       } else {
334           let mut r = pre[(-n - 1) as usize / 2];
335           r.y.negate_in_place(1);
336           r
337       }
338   }
```

However, it is implemented incorrectly:

```
373  fn table_verify(n: i32, w: usize) -> bool {
374      let n = n as usize;
375
376      ((n & 1) == 1) || (n >= !(1 << ((w - 1) - 1))) || (n <= (1 << ((w - 1) - 1)))
377  }
```

This definition contains several issues:

1. Incorrect cast to the `usize` type

The cast from **`i32`** to the **`usize`** type appears to serve as the **`abs(n)`** expression. However, casting a negative value reinterprets bits as large unsigned integer (two's complement), not an absolute value of **`n`**.

The intended range check should enforce symmetric bounds **`max`** and **`-max`** for the **`n`**.

2. Incorrect logical junctions

The predicates are combined with the **`||`** operator, but the **`&&`** operator must be used. All constraints must hold simultaneously.

3. Incorrect order of operations in bit shift

The intended limit is **`(1 << (w - 1)) - 1`**, but the current code computes **`1 << (w - 2)`**.

4. Incorrect arithmetic negation

The unary operator **`!`** is a bitwise "NOT", not arithmetic negation. It does not represent the negative bound, which should be symmetric to the positive bound.

5. Lack of robustness for allowed values of `w`

The current implementation relies on **`w`** staying within a narrow range. If **`w`** is ever changed to be outside this range, the shift or subtraction can underflow or become invalid for the chosen integer type:

- Underflow when **`w < 2`**
- Overflow when **`w > 31`**

The aforementioned flaws do not introduce incorrect values of wNAF digits, however they make the function **`table_verify`** significantly more permissive. This makes it not effective in preventing incorrect values of wNAF digits via debug assertions which is its only purpose.

This is especially important because there are no runtime checks in release mode due to performance reasons. If invalid wNAF digits reach table indexing expressions like **`pre[(n - 1) as usize / 2]`** or **`pre[(-n - 1) as usize / 2]`**, they can go out of bounds or select wrong points. This would panic in debug mode or produce incorrect scalar multiplication results in release mode.

Additionally, no equivalent validations of wNAF table digits have been identified in the corresponding secp256r1 implementation.

Recommendation

Consider translating the similar function in Bitcoin from C to Rust:

```
fn table_verify(n: i32, w: usize) -> bool {
    // `w` is constant, but this check makes the function future-proof
    // it would underflow/overflow when `w` is out of range
    debug_assert!((2..=31).contains(&w));

    // n must be odd
    if (n & 1) == 0 {
        return false;
    }

    let max: i32 = (1_i32 << (w - 1)) - 1;
    n >= -max && n <= max
}
```

Additionally, implement similar validation in **`secp256r1`**.

This issue has been resolved completely.

8 Miscellaneous validation issues

Severity: Low Impact: Medium Likelihood: Low Status: Addressed

Imprecise range check includes the modulus

In the `8x32` backend, the `from_bytes` function performs an incorrect range check when validating field elements parsed from raw bytes:

```
crypto/src/secp256k1/field_8x32.rs
pub(super) fn from_bytes
58    let value = Self::from_bytes_unchecked(bytes);
59
60    if u256::leq(&value.0, &Self::MODULUS.0) {
61        Some(value)
62    } else {
63        None
64    }
```

This condition `u256::leq` accepts values equal to P, whereas valid field elements must satisfy $0 \leq x < P$. As a result, the modulus P itself is incorrectly accepted as a valid field element.

Missing defensive magnitude check in `normalize_in_place`

The `normalize_in_place` function assumes that magnitude of the internal field element `self.magnitude` does not exceed `31`, but this invariant is not enforced locally:

```
crypto/src/secp256k1/field/field_10x26.rs
pub(super) const fn normalize_in_place
559   // Reduce self.0[9] at the start so there will be at most a single carry from the first pass
560   let mut x = self.0[9] >> 22;
561   self.0[9] &= 0x03FFFFF;
```

If `self.magnitude > 31`, intermediate arithmetic could overflow, leading to incorrect field normalization. A debug assertion should ensure this assumption.

In the current version of the codebase, this condition is not reachable: the maximum observed magnitude at this point is `10`. However, future changes (e.g., repeated add() or negate() operations) could introduce code paths where the magnitude exceeds `31`.

Missing defensive overflow checks

```
crypto/src/secp256k1/scalars/scalar64.rs
fn muladd(a: u64, b: u64, c0: u64, c1: u64, c2: u64)
316   let t = (a as u128) * (b as u128);
317   let th = (t >> 64) as u64; // at most 0xFFFFFFFFFFFFFFFE
318   let tl = t as u64;
319
320   let (new_c0, carry0) = c0.overflowing_add(tl);
321   let new_th = th.wrapping_add(carry0 as u64); // at most 0xFFFFFFFFFFFFFFFF
322   let (new_c1, carry1) = c1.overflowing_add(new_th);
323   let new_c2 = c2 + (carry1 as u64);
324
325   (new_c0, new_c1, new_c2)
```

Similarly in:

- **`crypto/src/secp256k1/scalars/scalar64.rs`**
  - **`muladd_fast`** on lines 328–340
  - **`sumadd_fast`** on lines 342–347
- **`crypto/src/secp256k1/scalars/scalar32.rs`**
  - Corresponding 32-bit implementations

Unenforced input bound in `sub_mod_with_carry`

The **`sub_mod_with_carry`** function relies on the precondition $a < 2 \cdot P$, but this bound is not enforced. If the invariant is violated, the result is not guaranteed to lie in $[0, P)$.

```
crypto/src/bigint_delegation/u256.rs
142   /// Note: we assume `self < 2*modulus`, otherwise the result might not be in the range
143   /// # Safety
144   /// `DelegationModParams` should only provide references to mutable statics.
145   /// It is the responsibility of the caller to make sure that is the case
146   pub unsafe fn sub_mod_with_carry<T: DelegatedModParams<4>>(a: &mut U256, carry: bool) {
147       let borrow = delegation::sub(a, T::modulus()) != 0;
148
149       if borrow && !carry {
150           delegation::add(a, T::modulus());
151       }
152   }
```

Same issue in **`crypto/src/bigint_delegation/u512.rs`**.

In the current version of the codebase, all callers respect the required bounds.

Unsafe buffer bounds assumption

Invoking the function **`assert_unchecked`** is immediate Undefined Behavior if the condition does not actually hold:

```
crypto/src/blake2s/delegated_extended.rs
fn finalize_impl
unsafe {
    // write zeroes
    let start = self
        .state
        .input_buffer
        .as_mut_ptr()
        .cast::<u8>()
        .add(self.buffer_filled_bytes);
    let end = self.state.input_buffer.as_mut_ptr_range().end.cast::<u8>();
    core::hint::assert_unchecked(start <= end);
    core::ptr::write_bytes(start, 0, end.offset_from_unsigned(start));
    // and run round function
    self.state
        .run_round_function_with_byte_len::<false>(self.buffer_filled_bytes, true);

    core::mem::transmute_copy::<_, [u8; 32]>(self.state.read_state_for_output_ref())
}
```

If **`start <= end`** is not guaranteed to hold and this can be influenced by user input, then **`assert_unchecked`** must be removed.

Even if it can be proven, add **`debug_assert!(start <= end)`** to catch invariant breaks during test suite runs or fuzzing.

Unchecked internal invariant in pairing Miller loop

Deterministic panic if curve configuration constants and precomputation logic diverge.

In BLS12-381:

```
crypto/src/bls12_381/curves/pairing_impl.rs
fn multi_miller_loop
120   for i in BitIteratorBE::without_leading_zeros(Config::X).skip(1) {
121       f.square_in_place();
122       Self::ell(&mut f, &ell_coeffs.next().unwrap(), &p.0);
123       if i {
124           Self::ell(&mut f, &ell_coeffs.next().unwrap(), &p.0);
125       }
126   }
```

In BN254:

```
crypto/src/bn254/curves/pairing_impl.rs
fn multi_miller_loop
115   let mut ell_coeffs = q.ell_coeffs.iter();
116
117   for i in (1..Config::ATE_LOOP_COUNT.len()).rev() {
118       if i != Config::ATE_LOOP_COUNT.len() - 1 {
119           f.square_in_place();
120       }
121
122       Self::ell(&mut f, ell_coeffs.next().unwrap(), &p.0);
123
124       let bit = Config::ATE_LOOP_COUNT[i - 1];
125       if bit == 1 || bit == -1 {
126           Self::ell(&mut f, &ell_coeffs.next().unwrap(), &p.0);
127       }
128   }
```

Validate `ell_coeffs.len()` against `EXPECTED_ELL_COEFFS` using a debug assertion. This would improve diagnosticability.

The range check in the `from_bytes` function has been corrected to exclude the field modulus P. Defensive magnitude check and overflow checks have been implemented.

9 Concurrent usage is not supported

Severity: QA Impact: Low Likelihood: Low Status: Notified

The library is designed using global static variables that must be initialized before calling the functions defined by the library:

```
crypto/src/lib.rs
pub fn init_lib
82    bn254::fields::init();
83    bls12_381::fields::init();
84    secp256k1::init();
85    bigint_delegation::init();
86    secp256r1::init();
```

The crates themselves contain multiple global static variables:

```
crypto/src/bigint_delegation/u256.rs
 7    static mut COPY_PLACE_0: MaybeUninit<U256> = MaybeUninit::uninit();
 8    static mut COPY_PLACE_1: MaybeUninit<U256> = MaybeUninit::uninit();
 9    static mut COPY_PLACE_2: MaybeUninit<U256> = MaybeUninit::uninit();
10    static mut COPY_PLACE_3: MaybeUninit<U256> = MaybeUninit::uninit();
11    static mut ONE: MaybeUninit<U256> = MaybeUninit::uninit();
12    static mut ZERO: MaybeUninit<U256> = MaybeUninit::uninit();
13    static mut SCRATCH: MaybeUninit<U256> = MaybeUninit::uninit();
```

Such declarations have been identified in modules **`secp256k1`**, **`secp256r1`**, **`bigint_delegation`**, **`bls12_381`** and **`bn254`**.

Beyond the requirement to initialize the library before use, a more significant consequence of using global static variables is that the library cannot be safely used concurrently or in a multi-threaded environment.

Usage within ZKsync OS has been analyzed and is safe since no concurrency is intended. Additionally, some test cases in the test suite are disabled with the comment **`"requires single threaded runner"`**.

However, this limitation should be kept in mind by any third-party developers building on top of the **`crypto`** library. Not only multi-threaded execution, but also concurrent usage may silently cause data races and potentially unsafe behavior if accessed concurrently without external synchronization.

Such usages could be:

- Async runtimes running on a single OS thread but interleaving execution
- Interrupt-driven execution

If the library needs to be used concurrently by community developers, it should be forked and modified to include synchronization.

Recommendation

Document the initialization and concurrency constraints for using the library for potential community developers.

10 Variable-time scalar operations should not be used with secret data

Severity: QA Impact: Low Likelihood: Low Status: Addressed

Several functions involved in core cryptographic operations are not constant-time. If these functions are ever used with secret inputs, attackers may exploit timing variability to gather statistical information and potentially recover secret data.

This observation is relevant for both **`secp256k1`** and **`secp256r1`** modules. See the "Code Duplicates" issue for identified instances of functions **`mul_shift_384_vartime`**.

Function `mul_shift_384_vartime` and scalar decomposition

The argument-dependent timing of the **`mul_shift_384_vartime`** function comes from the conditional branch at the end. The variable **`l`** is derived from the parameter **`b`**. The **`if`** statement decides whether to execute **`self.add_in_place(&Self::ONE)`**based on that bit. This creates data-dependent control flow: one path does the addition, the other does not.

src/secp256k1/scalars/scalar64.rs

```
149   pub(super) fn mul_shift_384_vartime(&mut self, b: &Self)
150   let words = b.0.as_words();
151   let l = words[1];
152   self.0 = U256::from_words([words[2], words[3], 0, 0]);
153
154   if (l >> 63) & 1 != 0 {
155       self.add_in_place(&Self::ONE);
156   }
```

The function is used in the **`decompose`** function, which implements the scalar decomposition operation, which in its turn affects the scalar multiplication operation.

Function `pow_vartime`, scalar inversion and affine conversion

Scalar inversion, implemented by the **`invert_assign`** function, uses the function **`pow_vartime`** which performs variable-time exponentiation, creating potential timing side-channels:

src/secp256r1/scalar/mod.rs

```
pub fn pow_vartime(&self, exp: &[u64])
 86    while j > 0 {
 87        j -= 1;
 88        res.square_assign();
 89
 90        if ((exp[i] >> j) & 1) == 1 {
 91            res.mul_assign(self);
 92        }
 93    }
```

Timing variability affects the functions relying on scalar inversion:

- **`invert_assign`** calls **`pow_vartime`** in **`src/secp256r1/scalar/mod.rs:68`**
- **`verify`** calls **`invert_assign`** in **`src/secp256r1/verify.rs:25`**

In the current codebase, these variable-time functions operate only on public inputs (signature scalars, message hashes, public keys). No secret material flows through them, so there is no concrete side-channel on secret keys. For secp256k1, the README also explicitly states that the module is intended solely for ecrecover-style verification. However, secp256r1 provides no equivalent usage guarantee. The primary risk is future misuse — these functions must not be reused in signing or other secret-key operations without replacing them with constant-time implementations.

11 Normalization check is not strong enough

Severity: QA Impact: Low Likelihood: Low Status: Notified

In module `secp256k1`, the function `normalize_in_place` is defined, conditioning on values of the `normalized` flag and `magnitude` tracker:

```
crypto/src/secp256k1/field/field_impl.rs
pub(super) fn normalize_in_place
129   if !self.normalized || self.magnitude > 1 {
130       self.value.normalize_in_place();
131       self.magnitude = 1;
132       self.normalized = true;
133   }
```

However, this condition has a minor logical flaw — it assumes that it is potentially possible that `self.normalized` is true but `self.magnitude` is not `1`. When a value is normalized, its magnitude must always be `1`.

Recommendation

Replace this code with:

```
if self.normalized {
    debug_assert!(self.magnitude == 1);
} else {
    debug_assert!(self.magnitude > 1);

    self.value.normalize_in_place();
    self.magnitude = 1;
    self.normalized = true;
}
```

12 Unit tests issues

Severity: QA Impact: Low Likelihood: Low Status: Addressed

Outdated instructions in the `README`

Running the tests as described in the **`README`** results in several failures, all throwing error **`ZKsync OS bin file missing: /.../zksync_os/for_tests.bin"`**.

Examples of such test cases:

- **`test_0_gas_limit`**
- **`fibish_sol`**

After inspecting the project's CI workflows, we have discovered that the script **`dump_bin.sh`** should be executed with **`--type for-tests`** flag now, in order to include new test cases into the coverage:

```
./dump_bin.sh --type for-tests
```

Some of the unit tests are not executed during CI

The following modules are not executed during regular CI runs:

- **`secp256k1::field::field_8x32`**
- **`secp256k1::scalars::scalar32_delegation`**

Additional workflow should be created that runs the test suite in single-threaded mode, filtering only those modules that require this:

```
cargo test --release -p crypto secp256k1::field::field_8x32 -- --test-threads=1 --ignored
cargo test --release -p crypto secp256k1::scalars::scalar32_delegation -- --test-threads=1 --
ignored
```

Some of the unit tests require larger stack

The test case **`secp256k1::field::field_8x32::tests::test_invert`** overflows the stack of standard size (8MB):

```
$ cargo test -p crypto secp256k1::field::field_8x32::tests::test_invert -- --test-threads=1 --
ignored
...
test secp256k1::field::field_8x32::tests::test_invert ...
thread '<unknown>' has overflowed its stack
fatal runtime error: stack overflow, aborting
```

It does succeed in case of bigger stack:

```
$ RUST_MIN_STACK=33554432 cargo test -p crypto secp256k1::field::field_8x32::tests::test_invert --
--test-threads=1 --ignored
...
running 1 test
test secp256k1::field::field_8x32::tests::test_invert ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 110 filtered out; finished in 0.57s

     Running tests/secp256k1.rs
```

Insufficient Test Coverage in `callable_oracles` Crate

- The **`callable_oracles`** crate has
  - no dedicated fuzzing targets
  - minimal Unit Testing:
    - Only 1 unit test in the entire **`callable_oracles`** crate (**`try_compute_hash_to_prime`**)
    - No property-based tests
    - No integration tests

Limitations of Fuzzing Coverage in `crypto` Crate

- The **`crypto`** crate has only one fuzzing gap:
  - BLS12-381:
    - Only 1 proptest regression
    - No dedicated fuzz targets

14 Excessive magnitude increment

Severity: QA Impact: Low Likelihood: Low Status: Notified

In the `secp256k1` module, the function `add_int_in_place` overestimates the magnitude increase of a field element:

```
src/secp256k1/field/field_impl.rs
pub(super) fn add_int_in_place(&mut self, rhs: u32)
114     self.magnitude += rhs;
115     debug_assert!(self.magnitude <= Self::max_magnitude());
116
117     self.value.add_int_in_place(rhs);
118     self.normalized = false;
```

Here, the field element is incremented by a small scalar `rhs`. However, the correct magnitude increment should be `ceil(rhs / radix)`, where the radix depends on the limb representation: 2^52 for the 5×52 backend, 2^26 for 10×26 backend, etc.

In practice, for the 5×52 representation, the radix is large enough that the increment is always exactly `1`, even for the maximum `u32` input. For the 10×26 representation, the correct increment may range from `1` up to `2^(32 - 26) = 64`, depending on how close `rhs` is to `u32::MAX`.

This issue does not affect correctness or security; it only potentially reduces performance.

Recommendation

Adjust the magnitude update so that the increment is always `1` in the 5×52 backend, and between `1` and `64` (inclusive) in the 10×26 backend, depending on the value of `rhs`.

15 Magic numbers decrease maintainability

Severity: QA Impact: Low Likelihood: Low Status: Addressed

Throughout the codebase, hard-coded number literals without context or a description are used. Using such “magic numbers” goes against best practices as they reduce code readability and maintenance as developers are unable to easily understand their use and may make inconsistent changes across the codebase.

Instances of magic numbers have been discovered in the following locations.

File `crypto/src/secp256k1/field/field_impl.rs`

Hard-coded literal `8` is used as the limit for magnitudes.

```
69    pub(super) fn mul_in_place(&mut self, rhs: &Self) {
70        debug_assert!(self.magnitude <= 8);
71        debug_assert!(rhs.magnitude <= 8);
72
73        self.value.mul_in_place(&rhs.value);
74        self.magnitude = 1;
75        self.normalized = false;
76    }
```

```
86    pub(super) fn square_in_place(&mut self) {
87        debug_assert!(self.magnitude <= 8);
```

```
150   pub(super) const fn mul(&self, rhs: &Self) -> Self {
151       debug_assert!(self.magnitude <= 8);
152       debug_assert!(rhs.magnitude <= 8);
```

```
166   pub(super) const fn square(&self) -> Self {
167       debug_assert!(self.magnitude <= 8);
```

File `crypto/src/secp256k1/points/affine.rs`

Hard-coded literal **`32`** is used as the limit for magnitudes, but also it is redundant code since **`Self::X_MAGNITUDE_MAX`** and **`Self::Y_MAGNITUDE_MAX`** are defined as a lesser number **`4`**.

```
crypto/src/secp256k1/points/affine.rs
41    pub(crate) const fn assert_verify(&self) {
42        #[cfg(all(debug_assertions, not(feature = "bigint_ops")))]
43        {
44            debug_assert!(self.x.0.magnitude <= Self::X_MAGNITUDE_MAX);
45            debug_assert!(self.x.0.magnitude <= 32);
46            debug_assert!(self.y.0.magnitude <= Self::Y_MAGNITUDE_MAX);
47            debug_assert!(self.y.0.magnitude <= 32);
48        }
49    }
```

Same code snippet is also implemented in lines 122-130 in the same file.

File `crypto/src/secp256k1/points/jacobian.rs`

Similarly, same literal **`32`** is hard-coded and also has no effect because of stronger equations involving **`X_MAGNITUDE_MAX`**, **`Y_MAGNITUDE_MAX`** and **`Z_MAGNITUDE_MAX`**:

```
crypto/src/secp256k1/points/jacobian.rs
28    pub(super) const fn assert_verify(&self) {
29        #[cfg(all(debug_assertions, not(feature = "bigint_ops")))]
30        {
31            debug_assert!(self.x.0.magnitude <= Self::X_MAGNITUDE_MAX);
32            debug_assert!(self.x.0.magnitude <= 32);
33            debug_assert!(self.y.0.magnitude <= Self::Y_MAGNITUDE_MAX);
34            debug_assert!(self.y.0.magnitude <= 32);
35            debug_assert!(self.z.0.magnitude <= Self::Z_MAGNITUDE_MAX);
36            debug_assert!(self.z.0.magnitude <= 32);
37        }
38    }
```

Same code snippet is also implemented in lines 195-205 in the same file.

Impact

Using magic numbers leads to maintainability issues and confusion, especially when they need to be updated or dynamically configured.

Recommendation

Declare named constants or implement configuration values instead of directly embedding literals to improve clarity and reduce potential errors in future development.

16 Code duplicates

Severity: QA Impact: Low Likelihood: Low Status: Notified

In the codebase, several code duplicates have been discovered:

- The function **`bits_var`** has several exact copies:
  - **`crypto/src/secp256r1/scalar/mod.rs:132-144`**
  - **`crypto/src/secp256k1/scalars/scalar32_delegation.rs:169-184`**
  - **`crypto/src/secp256k1/scalars/scalar64.rs:94-107`**
  - Additionally, there is one slightly modified copy:
  - **`crypto/src/secp256k1/scalars/scalar32.rs:92-104`**
- The function **`mul_shift_384_vartime`** is duplicated in:
  - **`crypto/src/secp256k1/scalars/scalar64.rs:145-156`**
  - **`crypto/src/secp256k1/scalars/scalar32_delegation.rs:228-239`**
  - **`crypto/src/secp256k1/scalars/scalar32.rs:126-138`**
- The affine equality **`eq`** is also duplicated in:
  - **`src/secp256r1/points/affine.rs:82`**
  - **`src/secp256k1/points/affine.rs:81`**
  - **`src/secp256k1/points/affine.rs:249`**

Code duplication increases security risks, as it hinders maintainability and also reviewability. Additionally, all future issue fixes must be manually applied and reviewed in all areas of duplication.

Recommendation

We recommend extracting common code into auxiliary functions in order to improve maintainability and reviewability of the codebase.

17 Redundant `else` branch

Severity: QA Impact: Low Likelihood: Low Status: Notified

In the function **`mul_assign`**, the "schoolbook" algorithm is implemented. Right after iterating **`j`** up to **`4`**, the **`if`** splits execution depending on the value of **`i + j`**:

crypto/src/secp256r1/scalar/scalar64.rs

```
pub(super) fn mul_assign
101     while j < 4 {
102         let k = i + j;
103
104         ...
```

```
114         j += 1;
115     }
116
117     if i + j >= 4 {
118         hi[i + j - 4] = carry;
119     } else {
120         lo[i + j] = carry;
121     }
```

If **`i + j < 4`**, the **`else`** branch would execute. However, this condition is never satisfied because after the **`while`** loop, **`j`** is guaranteed to equal **`4`**, and **`i`** is always greater than or equal to zero.

Appendix

Fuzzing parameters

The fuzzing was executed with the following parameters and targets:

- Duration
  - **48 hours**
- Machine:
  - CPU: Virtual AMD EPYC-Milan
    - **2 GHz, 48 Threads**
    - 32 MiB L3 Cache
  - RAM: 184 GiB

Fuzzing targets

`blake2s`	`bn254_ecadd`
`bn254_ecmul`	`bn254_pairing_check`
`ecrecover`	`p256_verify`
`keccak256`	`sha256`
`ripemd160`	`modexp`
`modexp_delegation`	`precompiles_ecadd`
`precompiles_ecmul`	`precompiles_ecpairing`
`precompiles_ecrecover`	`precompiles_p256`
`precompiles_sha256`	`precompiles_ripemd160`
`precompiles_modexp`	`precompiles_modexplen`

Fuzzing coverage

Generated using the command `./fuzz.sh coverage`.

Crate `callable_oracles`

File / Module	Lines Covered	Line Coverage	Functions Covered	Function Coverage
callable_oracles	114 / 650	17.5%	13 / 78	16.7%
src/arithmetic/mod.rs	53 / 54	98.1%	7 / 17	41.2%
src/hash_to_prime/common.rs	0 / 247	0.0%	0 / 34	0.0%
src/hash_to_prime/compute.rs	0 / 75	0.0%	0 / 5	0.0%
src/hash_to_prime/evaluate.rs	0 / 28	0.0%	0 / 2	0.0%
src/hash_to_prime/verify.rs	0 / 119	0.0%	0 / 5	0.0%
src/hash_to_prime/lib.rs	0 / 11	0.0%	0 / 2	0.0%
src/utils/evaluate.rs	42 / 96	43.8%	2 / 9	22.2%
src/utils/usize_slice_iterator.rs	19 / 20	95.0%	4 / 4	100.0%
Total	114 / 650	17.5%	13 / 78	16.7%

Crate `crypto`

File / Module	Lines Covered	Line Coverage	Functions Covered	Function Coverage
src/ark_ff_delegation	0 / 1080	0.0%	0 / 210	0.0%
src/biginteger/mod.rs	0 / 387	0.0%	0 / 74	0.0%
src/biginteger/const_helpers.rs	0 / 117	0.0%	0 / 19	0.0%
src/fp/mod.rs	0 / 414	0.0%	0 / 91	0.0%

File / Module	Lines Covered	Line Coverage	Functions Covered	Function Coverage
src/fp/montgomery_backend.rs	0 / 162	0.0%	0 / 26	0.0%
src/bigint_delegation/delegation.rs	0 / 82	0.0%	0 / 16	0.0%
src/bigint_delegation/mod.rs	0 / 4	0.0%	0 / 1	0.0%
src/bigint_delegation/u256.rs	0 / 209	0.0%	0 / 29	0.0%
src/bigint_delegation/u512.rs	0 / 189	0.0%	0 / 14	0.0%
src/blake2s/naive.rs	20 / 30	66.7%	19 / 29	65.5%
crypto/bls12_381	0 / 591	0.0%	0 / 65	0.0%
src/bls12_381/curves/g1.rs	0 / 92	0.0%	0 / 14	0.0%
src/bls12_381/curves/g2.rs	0 / 105	0.0%	0 / 11	0.0%
src/bls12_381/curves/pairing_impl.rs	0 / 209	0.0%	0 / 21	0.0%
src/bls12_381/curves/util.rs	0 / 166	0.0%	0 / 14	0.0%
src/bls12_381/fields/fq2.rs	0 / 13	0.0%	0 / 4	0.0%
src/bls12_381/fields/fq6.rs	0 / 6	0.0%	0 / 1	0.0%
crypto/bn254	0 / 306	0.0%	0 / 36	0.0%
src/bn254/curves/g1.rs	0 / 31	0.0%	0 / 7	0.0%
src/bn254/curves/g2.rs	0 / 26	0.0%	0 / 5	0.0%
src/bn254/curves/pairing_impl.rs	0 / 235	0.0%	0 / 22	0.0%
src/bn254/fields/fq2.rs	0 / 3	0.0%	0 / 1	0.0%
src/bn254/fields/fq6.rs	0 / 11	0.0%	0 / 1	0.0%
src/bn254/glv_decomposition.rs	0 / 93	0.0%	0 / 12	0.0%
src/bn254/lib.rs	0 / 2	0.0%	0 / 1	0.0%
src/bn254/raw_delegation_interface.rs	0 / 18	0.0%	0 / 2	0.0%

File / Module	Lines Covered	Line Coverage	Functions Covered	Function Coverage
crypto/secp256k1	1663 / 2105	79.0%	136 / 188	72.3%
src/secp256k1/mod.rs	0 / 34	0.0%	0 / 2	0.0%
src/secp256k1/context.rs	0 / 34	0.0%	0 / 4	0.0%
src/secp256k1/field/field_5×52.rs	332 / 384	86.5%	20 / 27	74.1%
src/secp256k1/field/field_impl.rs	88 / 144	61.1%	17 / 27	63.0%
src/secp256k1/field/mod.rs	114 / 169	67.5%	24 / 35	68.6%
src/secp256k1/field/mod_inv64.rs	321 / 336	95.5%	16 / 17	94.1%
src/secp256k1/points/affine.rs	80 / 111	72.1%	10 / 15	66.7%
src/secp256k1/points/jacobian.rs	170 / 300	56.7%	6 / 12	50.0%
src/secp256k1/points/storage.rs	7 / 7	100.0%	1 / 1	100.0%
src/secp256k1/recover.rs	218 / 229	95.2%	11 / 11	100.0%
src/secp256k1/scalars/invert.rs	89 / 89	100.0%	2 / 2	100.0%
src/secp256k1/scalars/mod.rs	33 / 42	78.6%	10 / 13	76.9%
src/secp256k1/scalars/scalar64.rs	211 / 226	93.4%	19 / 22	86.4%
crypto/secp256r1	0 / 904	0.0%	0 / 92	0.0%
src/secp256r1/context.rs	0 / 20	0.0%	0 / 3	0.0%
src/secp256r1/field/fe64.rs	0 / 173	0.0%	0 / 28	0.0%
src/secp256r1/field/mod.rs	0 / 68	0.0%	0 / 6	0.0%
src/secp256r1/mod.rs	0 / 9	0.0%	0 / 1	0.0%
src/secp256r1/points/affine.rs	0 / 42	0.0%	0 / 6	0.0%
src/secp256r1/points/jacobian.rs	0 / 203	0.0%	0 / 10	0.0%
src/secp256r1/points/storage.rs	0 / 7	0.0%	0 / 1	0.0%

File / Module	Lines Covered	Line Coverage	Functions Covered	Function Coverage
src/secp256r1/scalar/mod.rs	0 / 56	0.0%	0 / 8	0.0%
src/secp256r1/scalar/scalar64.rs	0 / 198	0.0%	0 / 18	0.0%
src/secp256r1/u64_arithmatic.rs	0 / 12	0.0%	0 / 3	0.0%
src/secp256r1/verify.rs	0 / 65	0.0%	0 / 4	0.0%
src/secp256r1/wnaf.rs	0 / 51	0.0%	0 / 4	0.0%
src/sha3/mod.rs	20 / 20	100.0%	9 / 23	39.1%
Total	1703 / 5633	30.2%	164 / 718	22.8%

Unit tests coverage

The unit test suite has been executed with the **`e2e_proving`** feature disabled since enabling it caused the suite to crash (see the description of this issue in the main part of the report). To collect coverage data, **`llvm-cov`** has been used.

Since the test suite of the **`crypto`** crate is built in "property testing" fashion, each new pass can result in slightly different coverage. This means that to compute realistic coverage, it is necessary to aggregate results from multiple runs. Multiple **`llvm-cov`** passes have been performed in parallel, on same machine used for fuzzing (48 cores). Each pass has been performed in single-threaded mode since some of the test cases require it:

```
# first, we ensure that all test code is compiled
cargo llvm-cov -p crypto --no-report -- --list

# bigger stack is needed for `field_8x32::tests::test_invert`
export RUST_MIN_STACK=33554432

seq 1 96 \
| xargs -P48 -n1 sh -c '
  cargo llvm-cov -p crypto --no-report -- --test-threads=1 --ignored || true
'

cargo llvm-cov report -p crypto --html
```

Following are the coverage statistics produced by the commands above.

Crate `callable_oracles`

Filename	Function Coverage	Line Coverage	Region Coverage
src/arithmetic/mod.rs	41.2%	98.1%	0.0%
src/hash_to_prime/common.rs	0.0%	0.0%	0.0%
src/hash_to_prime/compute.rs	0.0%	0.0%	0.0%
src/hash_to_prime/evaluate.rs	0.0%	0.0%	0.0%
src/hash_to_prime/verify.rs	0.0%	0.0%	0.0%
src/hash_to_prime/lib.rs	0.0%	0.0%	0.0%
src/utils/evaluate.rs	22.2%	43.8%	0.0%
src/utils/usize_slice_iterator.rs	100.0%	95.0%	0.0%
Total	16.7%	17.5%	0.0%

Crate `crypto`

Filename	Function Coverage	Line Coverage	Region Coverage
crypto/src/ark_ff_delegation/biginteger/mod.rs	29.73% (22/74)	23.92% (94/393)	24.40% (133/545)
crypto/src/ark_ff_delegation/const_helpers.rs	40.00% (10/25)	34.94% (58/166)	29.02% (74/255)
crypto/src/ark_ff_delegation/fp/mod.rs	54.95% (50/91)	52.87% (221/418)	50.98% (311/610)
crypto/src/ark_ff_delegation/fp/montgomery_backend.rs	65.52% (19/29)	54.45% (104/191)	60.43% (168/278)
crypto/src/bigint_delegation/delegation.rs	100.00% (15/15)	100.00% (83/83)	100.00% (169/169)
crypto/src/bigint_delegation/mod.rs	100.00% (1/1)	100.00% (4/4)	100.00% (4/4)

Filename	Function Coverage	Line Coverage	Region Coverage
crypto/src/bigint_delegation/u256.rs	87.88% (29/33)	89.87% (213/237)	89.04% (406/456)
crypto/src/bigint_delegation/u512.rs	100.00% (14/14)	98.43% (188/191)	98.70% (456/462)
crypto/src/blake2s/naive.rs	0.00% (0/9)	0.00% (0/30)	0.00% (0/43)
crypto/src/blake2s/test.rs	0.00% (0/2)	0.00% (0/18)	0.00% (0/26)
crypto/src/bls12_381/curves/g1.rs	25.00% (4/16)	20.16% (26/129)	12.12% (24/198)
crypto/src/bls12_381/curves/g1_swu_iso.rs	100.00% (1/1)	100.00% (6/6)	100.00% (12/12)
crypto/src/bls12_381/curves/g2.rs	0.00% (0/10)	0.00% (0/105)	0.00% (0/176)
crypto/src/bls12_381/curves/g2_swu_iso.rs	100.00% (1/1)	100.00% (6/6)	100.00% (12/12)
crypto/src/bls12_381/curves/pairing_impl.rs	66.67% (14/21)	80.56% (174/216)	85.32% (308/361)
crypto/src/bls12_381/curves/util.rs	0.00% (0/10)	0.00% (0/166)	0.00% (0/320)
crypto/src/bls12_381/fields/fq.rs	100.00% (26/26)	98.01% (246/251)	93.78% (422/450)
crypto/src/bls12_381/fields/fq2.rs	50.00% (2/4)	46.15% (6/13)	42.86% (6/14)
crypto/src/bls12_381/fields/fq6.rs	100.00% (1/1)	100.00% (6/6)	100.00% (7/7)
crypto/src/bls12_381/fields/fr.rs	78.95% (15/19)	84.29% (118/140)	85.61% (113/132)
crypto/src/bls12_381/fields/mod.rs	100.00% (1/1)	100.00% (4/4)	100.00% (4/4)
crypto/src/bn254/curves/g1.rs	40.00% (4/10)	29.85% (20/67)	25.51% (25/98)
crypto/src/bn254/curves/g2.rs	0.00% (0/5)	0.00% (0/26)	0.00% (0/41)

Filename	Function Coverage	Line Coverage	Region Coverage
crypto/src/bn254/curves/pairing_impl.rs	68.18% (15/22)	81.82% (198/242)	87.38% (374/428)
crypto/src/bn254/fields/fq.rs	100.00% (19/19)	97.06% (198/204)	92.41% (353/382)
crypto/src/bn254/fields/fq2.rs	100.00% (1/1)	100.00% (3/3)	100.00% (3/3)
crypto/src/bn254/fields/fq6.rs	100.00% (1/1)	100.00% (11/11)	100.00% (17/17)
crypto/src/glv_decomposition.rs	91.67% (11/12)	96.35% (132/137)	94.52% (207/219)
crypto/src/lib.rs	100.00% (1/1)	100.00% (9/9)	100.00% (6/6)
crypto/src/raw_delegation_interface.rs	0.00% (0/2)	0.00% (0/18)	0.00% (0/17)
crypto/src/secp256k1/context.rs	0.00% (0/4)	0.00% (0/34)	0.00% (0/45)
crypto/src/secp256k1/field/field_10×26.rs	0.00% (0/36)	0.00% (0/624)	0.00% (0/876)
crypto/src/secp256k1/field/field_5×52.rs	0.00% (0/37)	0.00% (0/448)	0.00% (0/562)
crypto/src/secp256k1/field/field_8×32.rs	70.00% (21/30)	82.42% (211/256)	77.27% (170/220)
crypto/src/secp256k1/field/field_impl.rs	0.00% (0/30)	0.00% (0/152)	0.00% (0/203)
crypto/src/secp256k1/field/mod.rs	0.00% (0/52)	0.00% (0/347)	0.00% (0/323)
crypto/src/secp256k1/field/mod_inv32.rs	0.00% (0/22)	0.00% (0/353)	0.00% (0/533)
crypto/src/secp256k1/field/mod_inv64.rs	0.00% (0/18)	0.00% (0/348)	0.00% (0/550)
crypto/src/secp256k1/mod.rs	25.00% (1/4)	9.76% (4/41)	5.97% (4/67)
crypto/src/secp256k1/points/affine.rs	0.00% (0/21)	0.00% (0/139)	0.00% (0/186)
crypto/src/secp256k1/points/jacobian.rs	0.00% (0/20)	0.00% (0/384)	0.00% (0/660)
crypto/src/secp256k1/points/storage.rs	0.00% (0/1)	0.00% (0/7)	0.00% (0/5)

Filename	Function Coverage	Line Coverage	Region Coverage
crypto/src/secp256k1/recover.rs	0.00% (0/24)	0.00% (0/411)	0.00% (0/543)
crypto/src/secp256k1/scalars/invert.rs	0.00% (0/3)	0.00% (0/107)	0.00% (0/136)
crypto/src/secp256k1/scalars/mod.rs	0.00% (0/30)	0.00% (0/137)	0.00% (0/148)
crypto/src/secp256k1/scalars/scalar32_delegation.rs	63.16% (24/38)	61.03% (166/272)	39.93% (119/298)
crypto/src/secp256k1/scalars/scalar64.rs	0.00% (0/28)	0.00% (0/247)	0.00% (0/746)
crypto/src/secp256r1/context.rs	0.00% (0/3)	0.00% (0/20)	0.00% (0/32)
crypto/src/secp256r1/field/fe32_delegation.rs	4.35% (1/23)	7.00% (7/100)	8.33% (10/120)
crypto/src/secp256r1/field/fe64.rs	0.00% (0/30)	0.00% (0/185)	0.00% (0/480)
crypto/src/secp256r1/field/mod.rs	0.00% (0/15)	0.00% (0/256)	0.00% (0/177)
crypto/src/secp256r1/mod.rs	50.00% (1/2)	30.77% (4/13)	21.05% (4/19)
crypto/src/secp256r1/points/affine.rs	0.00% (0/6)	0.00% (0/42)	0.00% (0/62)
crypto/src/secp256r1/points/jacobian.rs	0.00% (0/14)	0.00% (0/240)	0.00% (0/447)
crypto/src/secp256r1/points/storage.rs	0.00% (0/1)	0.00% (0/7)	0.00% (0/3)
crypto/src/secp256r1/scalar/mod.rs	0.00% (0/10)	0.00% (0/80)	0.00% (0/88)
crypto/src/secp256r1/scalar/scalar64.rs	0.00% (0/21)	0.00% (0/209)	0.00% (0/564)
crypto/src/secp256r1/scalar/scalar_delegation.rs	5.56% (1/18)	9.59% (7/73)	11.76% (10/85)
crypto/src/secp256r1/u64_arithmatic.rs	0.00% (0/3)	0.00% (0/12)	0.00% (0/18)
crypto/src/secp256r1/verify.rs	0.00% (0/6)	0.00% (0/120)	0.00% (0/208)
crypto/src/secp256r1/wnaf.rs	0.00% (0/4)	0.00% (0/51)	0.00% (0/69)
crypto/src/sha3/mod.rs	0.00% (0/5)	0.00% (0/20)	0.00% (0/34)
zksync_os_runner/src/lib.rs	0.00% (0/3)	0.00% (0/37)	0.00% (0/36)

Filename	Function Coverage	Line Coverage	Region Coverage
Total	31.41% (326/1038)	27.28% (2527/9262)	27.51% (3931/14288)

TARAN SPACE

Security Auditing, Research, and Advisory
for the Decentralized Web