TONCO CLAMM DEX v1.6

Security Assessment

January 16, 2026

Prepared for:
Alexander Pimenov
TONCO

Prepared by: **Elvis Skoždopolj and Nicolas Donboly**

Table of Contents

Project Summary

Contact Information

The following project manager was associated with this project:

Mary O'Brien, Project Manager
mary.obrien@trailofbits.com

The following engineering director was associated with this project:

Benjamin Samuels, Engineering Director, Blockchain
benjamin.samuels@trailofbits.com

The following consultants were associated with this project:

Elvis Skoždopolj, Consultant
elvis.skozdopolj@trailofbits.com

Nicolas Donboly, Consultant
nicolas.donboly@trailofbits.com

Project Timeline

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

Date	Event
September 11, 2025	Pre-project kickoff call
September 22, 2025	Status update meeting #1
September 29, 2025	Status update meeting #2
October 6, 2025	Status update meeting #3
October 14, 2025	Status update meeting #4
October 21, 2025	Delivery of report draft
October 21, 2025	Report readout meeting
December 1, 2025	Delivery of final comprehensive report
January 16, 2026	Completion of fix review

Executive Summary

Engagement Overview

TONCO engaged Trail of Bits to review the security of its CLAMM DEX. The CLAMM DEX is a concentrated liquidity automated market maker implemented on the TON blockchain, providing a decentralized exchange with capital-efficient liquidity provisioning similar to Uniswap v3.

A team of two consultants conducted the review from September 15 to October 20, 2025, for a total of 10 engineer-weeks of effort. Our testing efforts focused on identifying vulnerabilities in fee calculations, access control enforcement, the asynchronous message-passing architecture, multihop operations, liquidity management, and data validation, and on evaluating the implementation's mathematical equivalence with the Uniswap v3 reference implementation. With full access to source code and documentation, we performed static and dynamic testing of the codebase using automated and manual processes, including stateless differential fuzzing against Uniswap v3.

Observations and Impact

Trail of Bits identified three high-severity vulnerabilities that enable direct theft of funds from the protocol. An incorrect fee growth accumulator assignment causes liquidity providers to lose earned fees when burning positions (TOB-TONCO-1). The router contract contains two critical flaws allowing attackers to drain its proxy TON or TON balance through malicious payload construction (TOB-TONCO-6, TOB-TONCO-21). Additionally, we discovered six medium-severity issues, including one enabling access control bypass through address spoofing (TOB-TONCO-2), missing slippage protection enabling sandwich attacks (TOB-TONCO-7), the risk of price manipulation through empty pools (TOB-TONCO-8), and arithmetic overflow risks that can cause permanent jetton loss (TOB-TONCO-22).

The identified issues show a pattern of insufficient data validation at message boundaries, allowing for exploitation through crafted inputs. The router contract demonstrates weak validation with multiple pathways for fund theft through malformed multihop cells, unsupported operations, and negative amount calculations. The FunC and Tolk implementations show insufficient rigor in low-level operations, with missing zero-address checks, incorrect data structure serialization, and improper tick format handling.

The codebase exhibits good separation of concerns with operations isolated into dedicated files, and TON's asynchronous message-passing model provides natural protection against transaction ordering attacks. However, the system lacks robust testing practices, clean coding practices, event emission for monitoring critical state changes, and documentation; it also exhibits full centralization with unrestricted admin control.

Recommendations

Based on the codebase maturity evaluation and findings identified during the security review, Trail of Bits recommends that TONCO take the following steps before deploying the system to production:

- **Remediate the findings disclosed in this report.** These findings should be addressed through direct fixes or broader refactoring efforts. Any high-severity or medium-severity findings should be addressed immediately and checked against the original Algebra implementation to ensure the old implementation is not vulnerable to the same issues.
- **Improve the existing test suite and add stateful differential fuzzing.** The current test suite requires expansion to approach 100% code coverage across all contract functions and critical execution paths. Enhance the differential fuzzing infrastructure to include stateful testing that executes sequences of operations while maintaining pool state across multiple transactions. This stateful approach should validate that complex multistep user flows maintain equivalence with Uniswap v3 under realistic usage patterns, including multihop operations, position management flows, and edge cases involving empty pools and tick boundaries. The testing suite should comprehensively test the input validation of each message using both positive and adversarial test cases.
- **Add event emission and monitoring capabilities.** Implement event logging using Tolk's external log message functionality for all critical state-changing operations, including swaps, liquidity modifications, position transfers, and administrative actions. This will enable external indexers and monitoring systems to track protocol behavior and detect anomalies.
- **Establish documentation standards and architectural diagrams.** Add NatSpec-style documentation to all functions, create architectural diagrams showing component interactions and user flows, and document all role-based permissions and their authorized actions throughout the system. Retroactively write a system specification following the guidelines outlined in appendix F.
- **Complete development and conduct a comprehensive follow-up audit before deploying the system to production.** Given the number and severity of the findings we identified, TONCO should remediate all disclosed vulnerabilities, complete the implementation of the Factory and Automated Liquidity Manager (ALM) components, and significantly expand the test suite to include comprehensive coverage of all components, including newly developed ones. After these improvements are complete and the system has undergone internal testing, Trail of Bits recommends conducting another full security audit to validate the remediations and assess the security posture of the complete system before mainnet launch.

Finding Severities and Categories

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

EXPOSURE ANALYSIS

Severity	*Count*
High	3
Medium	6
Low	4
Informational	11
Undetermined	4

CATEGORY BREAKDOWN

Category	*Count*
Access Controls	2
Auditing and Logging	1
Configuration	2
Data Validation	18
Error Reporting	1
Timing	2
Undefined Behavior	2

Project Goals

The engagement was scoped to provide a security assessment of the TONCO CLAMM DEX. Specifically, we sought to answer the following non-exhaustive list of questions:

- Can users lose funds through improper fee calculations or accounting errors in liquidity position management?
- Are access controls properly enforced across pool operations, preventing unauthorized users from bypassing critical restrictions or manipulating pool state?
- Can malicious actors exploit the asynchronous message-passing architecture of TON to manipulate prices, extract value from liquidity providers, or front run legitimate transactions?
- Does the router contract properly validate multihop operations and protect against theft of jetton balances?
- Are liquidity minting and burning operations protected against slippage attacks and sandwich attacks that could result in unfavorable execution prices?
- Can the pool initialization and liquidity deposit flows be exploited to manipulate the initial price or extract value from early liquidity providers?
- Are there data validation issues in position NFT management, account operations, or pool state transitions that could lead to loss of funds or denial of service?
- Does the system properly handle edge cases such as zero-liquidity positions, empty pools, tick limit overflows, and malformed message payloads?
- Do the FunC and Tolk implementations maintain mathematical equivalence with the reference Uniswap v3 Solidity implementation across all core AMM operations?
- Can attackers exploit the four-position limit per account to grief legitimate users or deny service by depositing worthless or low-value positions?
- Can pool reserves be incorrectly updated in order to steal assets from another pool?
- Can race conditions between the message flows be exploited in order to extract value from the protocol?

Project Targets

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

TONCO

Repository	https://github.com/cryptoalgebra/algebra-ton-contracts
Version	`6e0b40195e83363acb6c5a0fea8a9366cdb40962`
Type	FunC, Tolk
Platform	TON

Project Coverage

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

Pool Core AMM Implementation
We conducted a manual review of the pool's concentrated liquidity mechanisms, focusing on the tick implementation, fee growth accounting, liquidity calculations, and swap execution logic. We examined whether the FunC and Tolk implementations maintain mathematical equivalence with the reference Uniswap v3 Solidity implementation, reviewing fixed-point arithmetic operations, fee accumulator updates across tick boundaries, and liquidity delta calculations. This analysis uncovered issues with fee growth tracking (TOB-TONCO-1), tick management (TOB-TONCO-5), price manipulation through empty pools (TOB-TONCO-8), arithmetic overflow risks (TOB-TONCO-22), and tick format corruption (TOB-TONCO-25).

Access Controls and Authentication
We analyzed access control enforcement across all operations, examining role-based permissions and the pool lock mechanism intended to restrict operations during critical state transitions. This review identified vulnerabilities allowing the pool lock to be bypassed through address spoofing (TOB-TONCO-2) and users to improperly route positions to the ALM subsystem (TOB-TONCO-19).

Router and Multihop Operations
We reviewed the router contract's jetton handling, payload construction, and multihop swap logic. We examined TON amount calculations, excess refund mechanisms, and payload validation for multihop operations. This analysis identified vulnerabilities allowing theft of the router's proxy TON balance (TOB-TONCO-6), jetton loss from malformed multihop cells (TOB-TONCO-12), jetton retention from unsupported operations (TOB-TONCO-13), and router TON drainage through negative amount calculations (TOB-TONCO-21).

Position Management and NFT Operations
We reviewed the `PositionNFT` and `Account` contract implementations, examining position minting, burning, and reforging operations, as well as enforcement of the four-position limit per account. This review uncovered issues with zero-liquidity position minting (TOB-TONCO-9), missing validation for critical addresses (TOB-TONCO-10), position minting through the router's account via multihop shortcuts (TOB-TONCO-11), and position limit enforcement failures (TOB-TONCO-24).

Slippage Protection and Transaction Timing
We analyzed the system's resistance to transaction ordering attacks, examining how TON's asynchronous message-passing architecture affects MEV resistance. We investigated

whether liquidity operations include adequate slippage protection and whether attackers could manipulate execution prices. This analysis revealed missing slippage protection for mint and burn operations (TOB-TONCO-7). We looked for ways in which race conditions could be exploited to claim more fees than expected, to perform a denial of service on user actions, to steal another user's position, or to cause a loss of a liquidity position for another user.

Data Validation and Edge Cases
We tested input validation across message handlers, examining how the system processes user-provided data and handles edge cases. This testing identified data validation issues in the timelock configuration (TOB-TONCO-3) and ALM message handling (TOB-TONCO-4), as well as unused return values (TOB-TONCO-15), missing existence checks (TOB-TONCO-16), incorrect excess receiver handling (TOB-TONCO-20), and incomplete exception handling (TOB-TONCO-23).

Differential Fuzzing against Uniswap v3
We developed a stateless fuzzing test suite using differential testing to verify mathematical equivalence with the Uniswap v3 Solidity reference implementation. Using Foundry's FFI feature, we executed the TONCO FunC implementations alongside Uniswap v3's Solidity implementations, comparing results for individual mathematical functions including price calculations (`getNextSqrtPriceFromAmount0RoundingUp`, `getNextSqrtPriceFromAmount1RoundingDown`), token amount deltas (`getAmount0Delta`, `getAmount1Delta`), swap step computation (`computeSwapStep`), liquidity calculation (`computeLiquidity`), and fee collection logic. This led to the discovery of TOB-TONCO-22 and TOB-TONCO-26. While the fee growth computation testing did not uncover an issue, this computation should be tested further, as indicated in TOB-TONCO-27. Each test function ran randomized test cases, validating that both implementations produce identical results for the same inputs. This stateless approach validated the correctness of individual mathematical operations in isolation (refer to the Automated Testing section for detailed methodology and results).

Coverage Limitations

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

- **The ALM and Factory contracts were not reviewed, as they were not implemented at the time of the audit.** Once implemented, they will add substantial additional functionality that will integrate with the existing pool and router contracts. Given the widespread changes their implementation will introduce, and the number of vulnerabilities we identified in the current codebase, we recommend conducting another comprehensive security audit after these components are implemented and all disclosed findings are remediated.

- **Our differential fuzzing approach validated mathematical equivalence only through stateless testing**, where each test case operates independently without maintaining state between operations. The TONCO team should enhance the testing suite to include stateful differential fuzzing that executes sequences of operations (swaps, mints, burns, and position modifications) while maintaining pool state across multiple transactions. Stateful fuzzing would provide stronger assurance that complex multistep user flows maintain equivalence with Uniswap v3 under realistic usage patterns.
- **We deprioritized some instances of bounce handling and failures stemming from user errors** (e.g., not supplying enough TON for full message execution). The TONCO team indicated these failures were designed intentionally, and we deprioritized them where they would result in limited loss only for the user that made the error.
- **Our review of the mathematical formulas was limited.** We compared them with the Uniswap v3 reference implementation and reviewed them for correctness, but due to the difference between EVM and TVM arithmetic handling, they should be further verified using stateless and stateful fuzz testing.

Automated Testing

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

Test Harness Configuration

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

Tool	Description	Policy
Foundry	Foundry fuzz testing was used via a custom setup that allowed for differential fuzzing between the Solidity Uniswap v3 implementation and the TONCO implementation of the shared mathematical functions.	Appendix D

Areas of Focus

Our automated testing and verification work focused on the following:

- Testing mathematical equivalence between Uniswap v3 and TONCO via differential fuzzing

Test Results

The results of this focused testing are detailed below.

Arithmetic Comparison with Uniswap v3

Property	Tool	Result
`getNextSqrtPriceFromAmount0RoundingUp` is equivalent to the Uniswap v3 implementation.	Foundry	**Passed**
`getNextSqrtPriceFromAmount1RoundingDown` is equivalent to the Uniswap v3 implementation.	Foundry	**Passed**
`getAmount0Delta` is equivalent to the Uniswap v3 implementation.	Foundry	TOB-TONCO-26
`getAmount1Delta` is equivalent to the Uniswap v3 implementation.	Foundry	TOB-TONCO-26

`computeSwapStep` is equivalent to the Uniswap v3 implementation.	Foundry	**Passed**
`computeLiquidity` is equivalent to the Uniswap v3 `getLiquidityForAmounts` implementation.	Foundry	TOB-TONCO-22
`computeSwapStep` cannot produce a negative `amountRemaining`.	Foundry	**Passed**
`computeSwapStep` cannot produce a negative fee amount.	Foundry	**Passed**

Codebase Maturity Evaluation

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

Category	Summary	Result
Arithmetic	The codebase arithmetic formulas are clear and, in most cases, follow the Uniswap v3 reference implementation. However, we discovered some notable differences, like a potential overflow in the max liquidity calculation (TOB-TONCO-22), a difference in rounding in the amount delta calculations (TOB-TONCO-26), and a potential overflow due to the use of signed integers and checked arithmetic (TOB-TONCO-27). It would be beneficial to adapt the stateless fuzz testing suite from Uniswap v3 in order to verify that the arithmetic behaves as expected, and to continue to differentially fuzz the project arithmetic against the Uniswap v3 implementation. The codebase uses explicit rounding directions in places where this is expected; however, issue TOB-TONCO-26 indicates that these rounding directions might not exactly match the reference implementation.	**Moderate**
Auditing	The contracts lack formal event emission mechanisms for monitoring critical state changes. While Tolk supports external log messages through `createExternalLogMessage` for emitting events to indexers, the contracts do not implement any logging mechanisms. The TONCO team has developed an off-chain indexer that monitors on-chain messages to track protocol activity; however, this indexer was out of scope for this audit and was not evaluated. The team would benefit from investigating the addition of events through Tolk's `createExternalLogMessage` functionality, which might provide more reliable monitoring capabilities for external systems. Additionally, comprehensive monitoring coverage should include administrative functions, such as router admin and arbiter operations, to ensure that all privileged actions	**Further Investigation Required**

	are properly tracked and auditable.	
Authentication / Access Controls	Access controls are generally well implemented throughout the codebase, with proper authorization checks in most critical operations. However, the findings revealed that access control enforcement is not systematic, leading to specific vulnerabilities allowing user-controlled inputs to bypass intended restrictions (TOB-TONCO-2, TOB-TONCO-19). The lack of comprehensive access control testing means similar issues could exist elsewhere. A systematic approach with dedicated test coverage for all privileged operations would strengthen this area.	**Moderate**
Complexity Management	The codebase demonstrates decent separation of concerns, with operations properly isolated into dedicated files. This architectural decision makes the overall system structure clean and maintainable. However, certain critical functions, especially `reforgeOperation`, contain excessive complexity that hinders analysis and increases the risk of bugs. Breaking down complex functions into smaller, focused components would improve code maintainability and reduce the likelihood that vulnerabilities are introduced. Additionally, the codebase makes extensive use of deeply nested struct compositions, which adds cognitive overhead when tracing data flow through the system. This pattern of nested structures can make it difficult to understand the full state transformation without careful analysis of multiple struct definitions across different files. It would be beneficial to evaluate how these nested structs could be simplified and otherwise refactored to reduce the complexity. Additionally, these data structures should be clearly documented.	**Weak**
Cryptography and Key Management	No cryptographic operations were present in the reviewed scope.	**Not Applicable**
Decentralization	The system is fully centralized, with the router admin and arbiter roles having unrestricted control over critical operations. While timelock mechanisms provide some transparency for code updates, both privileged roles can perform actions without meaningful constraints. This	**Weak**

	centralization creates significant trust assumptions and single points of failure. These roles must be secured behind multisignature wallets. It would be beneficial to create user-facing documentation that clearly outlines the risks of interacting with the system and the protections in place that mitigate this risk.	
Documentation	Documentation quality is insufficient for a production system of this complexity. The codebase lacks inline comments, NatSpec annotations, and external documentation. No architectural diagrams exist to explain system components and their interactions. Critical information about roles and their authorized actions is undocumented. The TONCO team should add comprehensive NatSpec documentation to all functions, especially user-facing ones, maintain up-to-date external documentation with architectural diagrams covering major user flows, and clearly define each role's authorized actions throughout the system.	**Weak**
Low-Level Manipulation	FunC and Tolk are inherently low-level languages requiring careful handling of cell and slice manipulation for data serialization. The audit revealed multiple critical issues with data structure handling that demonstrate insufficient rigor in low-level operations. The codebase incorrectly assigns fee growth accumulators between token types (TOB-TONCO-1), fails to validate critical addresses that could result in permanent fund loss (TOB-TONCO-10), and exhibits problems with tick data format interpretation during system evolution (TOB-TONCO-25). These findings indicate that the code correctly performs basic operations to extract and manipulate data structures, but it fails to validate that the extracted data is meaningful or to handle edge cases appropriately. The extensive use of deeply nested struct compositions compounds these issues, as errors in serialization or deserialization at any level can cascade through the system. The TONCO team should implement comprehensive validation at serialization boundaries, add assertions for critical invariants, and establish clear documentation for all data structure formats and their evolution patterns.	**Moderate**
Testing and	The existing test suite has low coverage and lacks	**Weak**

Verification	end-to-end testing to simulate production environments. During this audit, we developed a fuzzing test suite using differential testing against Uniswap v3 via Foundry's FFI feature (refer to the Automated Testing section for more details). Additional unit tests are needed throughout the codebase. Once the ALM and Factory components are implemented, end-to-end integration tests will be mandatory to verify the complete system behavior. The TONCO team should target as close to 100% test coverage as feasible. Additionally, the team should explore newcomer tools like BugMagnifier for transaction simulation to further strengthen the system's security posture against asynchronous execution flaws.	
Transaction Ordering	The TON blockchain architecture inherently prevents traditional front-running seen on EVM chains. Beyond one identified issue with sandwich attacks (TOB-TONCO-7), the system demonstrates adequate resistance to transaction ordering attacks. The asynchronous message-passing model of TON provides natural protections against MEV exploitation vectors common in other DeFi protocols.	**Moderate**

Summary of Findings

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

ID	Title	Type	Severity
1	Fee calculation mismatch in PositionNFT burn operation causes incorrect fee growth tracking for token1	Data Validation	High
2	owner_address can be spoofed to bypass the pool lock in the swapOperation function	Access Controls	Medium
3	timelock_delay cannot be updated	Data Validation	Low
4	ALM mint path sends ALM address instead of user address in message body	Data Validation	Undetermined
5	Occupied ticks guard blocks addition of liquidity to existing ticks and allows MAX_USER_TICKS overflow	Data Validation	Medium
6	Proxy TON balance of the router can be stolen	Data Validation	High
7	Missing slippage protection for mint and burn orders	Timing	Medium
8	price_sqrt can be manipulated by swapping through an empty pool	Timing	Medium
9	Zero-liquidity positions can be minted and deposited into an account	Data Validation	Informational
10	Missing zero address check in the reforgeOperation function can lead to loss of funds	Data Validation	Low
11	Multihop shortcut allows a position to be minted through the router contract's account	Data Validation	Informational

12	Malformed multihop cell can lead to jetton loss	Data Validation	**Medium**
13	Router will keep the jettons if the transfer notification contains an unsupported operation	Data Validation	**Low**
14	Pool reinitialization risks	Configuration	**Low**
15	Return value of modifyPosition is ignored inside burnPosition	Data Validation	**Informational**
16	Router does not check for pool existence	Data Validation	**Informational**
17	Lack of transparency in timelocked code updates	Auditing and Logging	**Informational**
18	Manual balance calculation instead of raw_reserve	Configuration	**Informational**
19	Users are able to route positions to the ALM	Access Controls	**Undetermined**
20	Excess TON refunded to the wrong address in swaps	Data Validation	**Informational**
21	The router's TON balance can be drained via negative amount calculation	Data Validation	**High**
22	Overflow in getMaxLiquidityForAmount0Precise can result in fund loss	Data Validation	**Medium**
23	Incomplete handling of swap exceptions	Error Reporting	**Informational**
24	Depositing more than four positions causes failure	Data Validation	**Informational**
25	Incorrect price caching corrupts legacy tick format	Data Validation	**Informational**

26	Rounding difference between Uniswap v3 and TONCO amount delta calculations	Data Validation	**Undetermined**
27	Potential overflow in fee growth computation	Undefined Behavior	**Undetermined**
28	Functions missing the impure specifier	Undefined Behavior	**Informational**

Detailed Findings

1. Fee calculation mismatch in PositionNFT burn operation causes incorrect fee growth tracking for token1

Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-TONCO-1
Target: `contracts/positionnft_tolk/operations/positionnft_burn.tolk`	

Description

The `burnNFT` function in the `PositionNFT` contract assigns `token1` fee growth using the `token0` accumulator. This causes incorrect tracking of fee growth for `token1` and will lead to miscalculated fee claims and distributions for `token1`.

The `growthInside0LastX128` and `growthInside1LastX128` state variables are per-position snapshots of the fee-growth accumulators for `token0` and `token1` inside the position's tick range, scaled by 2^{128}. On each position update (mint, burn, collect), the system refreshes these snapshots to the latest "inside" accumulators, so future owed fees are computed from the difference between the current accumulators and the stored snapshots. Assigning `growthInside1LastX128` from the `token0` accumulator sets an incorrect baseline for `token1`, so subsequent calculations take deltas against `token0`'s history instead of `token1`'s and may underpay or overpay users depending on pool state evolution.

```
/*Now lets update the pos */
position0.liquidity -= liquidity2Burn;
position0.growthInside0LastX128 = newFees.growthInside0LastX128;
position0.growthInside1LastX128 = newFees.growthInside0LastX128;
```

Figure 1.1: Incorrect `token1` fee growth assignment (`contracts/positionnft_tolk/operations/positionnft_burn.tolk#L77–L80`)

Exploit Scenario

Alice monitors a pool where `token0` and `token1` fee growth diverge and `token1` fee growth (`growthInside1LastX128`) significantly exceeds that of `token0`. She opens a position and triggers a burn when `growthInside0LastX128` and `growthInside1LastX128` differ substantially. Because the burn path records `token1` growth using the `token0` accumulator, the contract later calculates `token1` fee deltas from an incorrect baseline, allowing Alice to harvest `token1` fees that were not actually accrued to the position.

Recommendations

Short term, replace the second line of the `token1` fee growth assignment (highlighted in figure 1.1) with `position0.growthInside1LastX128 = newFees.growthInside1LastX128` so that `token1` fee growth is sourced from the `token1` accumulator.

Long term, add unit tests to validate that the fee accumulator fields are assigned consistently in all position life cycle operations.

2. owner_address can be spoofed to bypass the pool lock in the swapOperation function

Severity: **Medium**	Difficulty: **Low**
Type: Access Controls	Finding ID: TOB-TONCO-2
Target: `contracts/pool/operations/pool_swap.func`	

Description

An attacker can bypass the pool contract's swap lock by supplying a crafted `ownerAddress` in a swap message sent through the router contract. The pool contract incorrectly trusts this user-controlled field to determine if the lock can be overridden, enabling authorization bypass.

The pool's lock is a security feature intended to allow an admin (a pool controller or pool creator) to temporarily halt swaps (e.g., during an emergency). To allow the admin to perform administrative actions while the lock is active, the contract is designed to let them bypass this check. The bypass is triggered if the `ownerAddress` field in the swap message matches the admin's address. However, the router passes this `ownerAddress` directly from user-controlled input without validation. This allows any user to impersonate the admin by simply setting the `ownerAddress` in their swap transaction to the admin's address, which makes the `overrideLock` flag true and allows them to bypass the swap lock at will.

```
try {
    int overrideLock = equal_slice_bits(owner_address,
pool::roles.at(POOL_ROLE_CONTROLLER)) |
                       equal_slice_bits(owner_address,
pool::roles.at(POOL_ROLE_CREATOR));

    if (((pool::flags & FLAG_LOCK_MASK) != 0) & (~ overrideLock))  {
        throw (POOL_INACTIVE);
    }
```

Figure 2.1: Pool derives lock-bypass `overrideLock` from `owner_address`. (`contracts/pool/operations/pool_swap.func#L88–L94`)

Exploit Scenario

While the TON/USDT pool is locked for an upgrade, an attacker, Eve, bypasses the restriction. She crafts a jetton transfer to the router and sets the `ownerAddress` in the payload to the pool's controller address. The router forwards this spoofed address to the pool, causing its `overrideLock` check to pass. This allows Eve to trade against the locked pool, letting her arbitrage stale prices or otherwise act against the admin's intent.

Recommendations

Short term, update the router and pool contracts to use the authenticated `fromUser` address for lock overrides. In `router_receive.tolk`, the router must pass the `fromUser` address from the jetton `transfer_notification` to the pool, not the user-supplied `ownerAddress`. The pool must then use this trusted address for its `overrideLock` check.

Long term, never derive access controls from user-controlled payload data; instead, base access controls on authenticated message properties, like the verified sender's address.

3. timelock_delay cannot be updated

Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-TONCO-3
Target: `contracts/pool/*`	

Description

The `timelock_delay` state variable cannot be updated to a new value, so pool contracts always use the default value of 5 minutes.

The `timelock_delay` state variable defines the amount of time that needs to pass before the update to the pool contract configuration parameters and code can be committed, after this update process is started. This value is initially set to zero when the pool is deployed since this is a part of the pool `stateInit` data, as shown in figure 3.1:

```
cell pack_pool_data(slice jetton0_wallet, slice jetton1_wallet) inline_ref {
    slice self = my_address();
    cell empty_cell = begin_cell().end_cell();

    cell data = begin_cell()
        // ...
        .store_ref(begin_cell()
            // ...
            .store_ref(begin_cell()
                .store_uint(0, 64) // stores the timelock_delay
            .end_cell())
        .end_cell())
    .end_cell();

    return (data);
}

(cell, int) calculate_pool_state_init(slice jetton0_address, slice jetton1_address,
cell pool_code) inline {
    int order = (slice_hash(jetton0_address) <=> slice_hash(jetton1_address));

    cell pool_data = order > 0 ?
        pack_pool_data(jetton0_address, jetton1_address) :
        pack_pool_data(jetton1_address, jetton0_address);

    cell state_init = begin_cell()
        .store_uint(0, 2)
        .store_dict(pool_code)
        .store_dict(pool_data)
```

```
        .store_uint(0, 1)
    .end_cell();

    return (state_init, order);
}
```

Figure 3.1: The `stateInit` calculation in `pool_utils.func` (`contracts/pool/pool_utils.func#L63`)

However, the pool contract does not contain logic to update this value. Due to this, the default value of 5 minutes will always be used.

Exploit Scenario

Alice, a TONCO team member, wishes to update the `timelock_delay` state variable after a request from multiple liquidity providers. However, she is unable to do this due to the missing functionality and is forced to upgrade the code of the pool contract in order to add the functionality to change this configuration parameter.

Recommendations

Short term, add a function to update the `timelock_delay`.

Long term, improve the coverage of the test suite. Retroactively create a specification that will contain all of the functionality that each contract is expected to have, and use this specification to guide the creation of unit tests.

4. ALM mint path sends ALM address instead of user address in message body

Severity: **Undetermined**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-TONCO-4
Target: `contracts/pool/operations/pool_mint.func`	

Description

The ALM mint path constructs and sends a message to the ALM with `owner_addr` appended to the message body after first verifying that `owner_addr` equals the ALM address. As a result, the message encodes the ALM address rather than the end user's address. The user address is not included anywhere in the message body in this branch. If the ALM interprets the appended address as the beneficiary or position owner, it will attribute the minted position or subsequent actions to the ALM itself, not to the intended user. This behavior stems from the overloading of `owner_addr` as both a routing selector (to detect ALM flow) and an identity field, without a separate beneficiary field or validation.

```
if (equal_slice_bits(owner_addr, pool_arbiter_alm)) {
    dumpstr("  Funding ALM wallet!");
    body = body.store_slice(owner_addr);
    send_simple_message(0, pool_arbiter_alm, body.end_cell(),
MODE_CARRY_REMAINING_GAS);
    return ();
}
```

Figure 4.1: The ALM branch appends `owner_addr` (the ALM address) to the message body. (`contracts/pool/operations/pool_mint.func#L41–L46`)

This code path differs from the non-ALM path, which routes to the liquidity provider account using `owner_addr` and the order payload. In the ALM path, the order payload is not forwarded, and the message body contains the ALM address, preventing the ALM from reliably inferring the intended user.

Exploit Scenario

Alice, a legitimate user, initiates a mint through the router with `owner_addr` set to the ALM address, causing the pool to enter the ALM branch. The pool then sends a message to the ALM with `owner_addr` appended to the body, which equals the ALM address. The ALM treats this field as the beneficiary for minting or accounting, so it attributes the position to itself and captures the user's funded liquidity or position rights rather than assigning them to the user. The user cannot receive or manage the minted position because their address is absent from the message body.

Recommendations

Short term, have the ALM mint path append the end user beneficiary address (not `owner_addr` when it equals the ALM) to the ALM message body. This will prevent misattribution by ensuring the ALM acts for the intended user.

Long term, when porting the pool contracts from FunC to Tolk, define a typed ALM message schema with a mandatory beneficiary field. This will prevent the issue because the user address will always be required and unambiguous.

5. Occupied ticks guard blocks addition of liquidity to existing ticks and allows MAX_USER_TICKS overflow

Severity: **Medium**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-TONCO-5
Target: `contracts/pool/amm.func`	

Description

The mint path guard rejects any liquidity addition when the number of occupied ticks has reached the cap, regardless of whether the targeted ticks are already initialized. This prevents legitimate liquidity providers from adding liquidity to existing ticks when the counter equals the maximum. In addition, when the counter is just below the cap, at 19999, and both targeted ticks are uninitialized, the guard permits the operation and the subsequent state updates increment the counter twice, allowing it to exceed the cap and break the intended invariant.

```
{- Only for mints! We must limit them if the limit is reached -}
if (liquidity > 0) {
    dumpint("  Checking pool::occupied_ticks: ", pool::occupied_ticks);
    if (pool::occupied_ticks >= MAX_USER_TICKS) {
        return (RESULT_TOO_MANY_TICKS);
    }
}
```

Figure 5.1: Guard rejects all mints at the cap, without checking if ticks already exist. (contracts/pool/amm.func#L107–L113)

```
...
pool::occupied_ticks += lowExisted ? 0 : 1;
...
pool::occupied_ticks += upExisted ? 0 : 1;
```

Figure 5.2: State updates can increase the counter by two when both ticks are new. (contracts/pool/amm.func#L180–L192)

When the cap is reached, liquidity providers should be allowed to add liquidity to previously initialized ticks because doing so does not create new occupied ticks. Conversely, when the cap has not yet been reached, addition of liquidity to two uninitialized ticks should be blocked if the resulting total would exceed the cap.

Exploit Scenario

Bob, an attacker, monitors the occupied tick count and waits until it is one below the cap. He submits a mint that targets two uninitialized ticks, which passes the guard because the

count is below the cap. The function then initializes both ticks and increments the counter twice, causing the occupied tick count to exceed the cap. This breaks the invariant that the number of occupied ticks never exceeds the maximum. Separately, when the count equals the cap, a legitimate liquidity provider who targets two already initialized ticks is rejected by the guard, preventing liquidity increases that should be allowed.

Recommendations

Short term, add a check to the mint path guard for tick existence before it enforces the cap; it should allow liquidity to be added when both ticks are already initialized and reject it only when the operation would create more occupied ticks than the maximum.

Long term, add tests asserting that adding liquidity to existing ticks never fails due to the cap and that `occupied_ticks` never exceeds `MAX_USER_TICKS` after any mint.

6. Proxy TON balance of the router can be stolen

Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-TONCO-6
Target: `contracts/router_tolk/operations/router_send.tolk`	

Description

The proxy TON jetton balance held by the router contract can be stolen when the `ROUTER_FLAG_MULTIHOP_SHORTCUT` flag is set to true due to missing validation in the multihop shortcut path.

The protocol allows users performing a swap to attach an arbitrary multihop payload; this allows users to more easily perform additional actions after the swap. The `payToOperation` function of the router contract implements an optional shortcut if the receiver of either of the jettons is the router contract or the router's proxy TON jetton wallet.

```
if ((router.flags & ROUTER_FLAG_MULTIHOP_SHORTCUT) != 0) {
    if (coins.amount0 > 0) {
        if (payToBody.receiver0 == contract.getAddress()) {
            transferNotificationPayloadProcess(queryId, coins.amount0,
contract.getAddress(), coins.jetton0Address, payloadCell0, tonAmount0);
            coins.amount0 = 0;
        }
        if (payToBody.receiver0 == router.proxyTonWallet) {
            transferNotificationPayloadProcess(queryId, coins.amount0,
contract.getAddress(), router.proxyTonWallet, payloadCell0, tonAmount0);
            coins.amount0 = 0;
        }
    }
    if (coins.amount1 > 0) {
        if (payToBody.receiver1 == contract.getAddress()) {
            transferNotificationPayloadProcess(queryId, coins.amount1,
contract.getAddress(), coins.jetton1Address, payloadCell1, tonAmount1);
            coins.amount1 = 0;
        }
        if (payToBody.receiver1 == router.proxyTonWallet) {
            transferNotificationPayloadProcess(queryId, coins.amount1,
contract.getAddress(), router.proxyTonWallet, payloadCell1, tonAmount1);
            coins.amount1 = 0;
        }
    }
}
```

Figure 6.1: A snippet of the multihop shortcut logic in the router contract (`contracts/router_tolk/operations/router_send.tolk#L98–L127`)

However, in the latter case, the `proxyTonWallet` is added as the `sender_address` input to the `transferNotificationPayloadProcess` function even though the jetton wallet is not verified to be the router's proxy TON wallet. This allows a malicious user to drain the proxy TON balance of the `router` by exchanging a less valuable or malicious jetton for proxy TON.

Exploit Scenario

The protocol holds $100,000 of proxy TON. Eve deploys two malicious jetton contracts, jetton A and jetton B, and uses the pool factory contract to deploy and set up a pool for these jettons. She swaps jetton A for jetton B, sets the receiver of jetton B to the router's proxy TON wallet address, and includes a multihop swap payload that will intentionally fail and trigger the jetton refund flow. Due to the missing validation, the refund returns proxy TON instead of jetton B, allowing Eve to steal all the proxy TON held by the router.

Recommendations

Short term, for both receiver checks shown in figure 6.1, add a check that, if the receiver is the `proxyTonWallet`, the jetton wallet used in the swap is also the `proxyTonWallet`.

Long term, improve the coverage of the test suite by adding tests for common adversarial scenarios and malicious or malformed data.

7. Missing slippage protection for mint and burn orders

Severity: **Medium**	Difficulty: **Medium**
Type: Timing	Finding ID: TOB-TONCO-7
Target: `contracts/common/reforge_message.tolk`	

Description

Mint and burn orders do not carry user-defined slippage bounds or a deadline, and the pool executes them against the current pool price without validating user intent. The mint order structure includes only the operation type, liquidity, tick bounds, and receiver, while the burn side carries only a liquidity-to-burn value.

Because these flows execute asynchronously across multiple messages, an attacker can change the price between the inclusion and execution of the order and force unfavorable fills, denial of service, or value extraction.

```
struct MintOrder {
    op : int32;
    liquidity: uint128;
    tickLower: int24;
    tickUpper: int24;
    nftReceiver : address;
}
```

Figure 7.1: Mint order schema has no minimum amount or deadline fields. (contracts/common/reforge_message.tolk#L88–L98)

```
struct BurnOrder {
    index    : uint64,
    subindex : uint4,
    data: PosAndFeesData,
    liquidity2Burn : int128
}
```

Figure 7.2: Burn order schema has no minimum amount fields to protect withdrawals. (contracts/common/reforge_message.tolk#L53–L75)

Exploit Scenario

Bob, an attacker, monitors newly included blocks for reforge transactions that include burn orders and initiates a large swap to move the pool price to a less favorable level for the position's tick range. He times the message execution so that the swap lands right before the pool executes the burn. The pool then processes the burn at the manipulated price and pays the user out in an imbalanced or reduced-value mix of tokens because no minimum

output or deadline is enforced to validate user intent. Immediately after the burn order is executed, Bob reverses the price move and captures the difference via arbitrage across the pool and external markets, extracting value from the user's withdrawal. The transaction cannot revert due to missing bounds.

Recommendations

Short term, add per-order slippage bounds to the mint and burn schemas and enforce them during execution so that orders revert when current amounts violate user thresholds.

Long term, add tests that simulate price movement between order creation and execution, and assert that mint and burn orders revert when bounds are exceeded and succeed when values are within bounds.

8. price_sqrt can be manipulated by swapping through an empty pool	
Severity: **Medium**	Difficulty: **Low**
Type: Timing	Finding ID: TOB-TONCO-8
Target: `contracts/pool/operations/pool_swap.func`	

Description
A malicious user can perform a swap through an empty pool in order to move the `price_sqrt` state variable to any value, extracting value from legitimate liquidity deposits.

When the pool is first deployed and initialized, the liquidity in the pool is zero. Since the swap operation does not have a check for the amount of total liquidity, swaps can be performed through an empty pool. The calculation performed for the swap will return 0 and refund the user the amount they used to perform the swap; however, the state changes made in the pool will be preserved. This allows a malicious actor to move the `price_sqrt` state variable to any value. If this is done prior to a legitimate liquidity deposit, the malicious actor could perform a swap at a new price that favors them, extracting value from the pool and the liquidity provider.

While front-running is usually not an issue on TON, in this case, the add liquidity message chain and the swap message chain require a different number of messages to execute before the pool state is modified. Specifically, starting from the transfer notification made to the router, it takes only one more message to execute the swap, while the add liquidity chain requires three more. Due to this difference, a malicious user can reliably execute their swap before the initial liquidity addition is made as soon as the liquidity addition transaction is included in a block.

Exploit Scenario
Alice, an honest user, deploys an unlocked pool and sets the initial `price_sqrt` so that 1 jetton A can be swapped for 1 jetton B. She starts the liquidity deposit by sending both jettons to the router. Eve notices this transaction and executes a swap before the liquidity addition has finalized, moving the `price_sqrt` variable to a different value. Eve then swaps jetton A for jetton B, receiving more jetton B than she normally would have had the price stayed the same.

Recommendations
Short term, consider preventing swaps through an empty pool. Otherwise, document this issue in the user-facing documentation and encourage pool creators to deploy locked pools

and to perform the initial liquidity addition before unlocking them. Add slippage checks that run when liquidity is minted.

Long term, have pools deployed as locked by default in the pool factory and allow users to configure whether their pools will be deployed as locked or unlocked. Add tests that execute operations outside of their intended order to more easily discover similar cases.

9. Zero-liquidity positions can be minted and deposited into an account

Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-TONCO-9
Target: `contracts/positionnft_tolk/operations/positionnft_burn.tolk`	

Description

The deposit flow permits zero- or partial-liquidity deposits without validation, which can cause position NFTs to be minted with zero liquidity. The `depositNFT` entrypoint accepts a deposit amount and forwards it as a burn-like order without enforcing a nonzero constraint. The account then executes the order if its conditions are satisfied and triggers a reforge. During a reforge, the pool may pass the position through and mint a new position whose `liquidity` field equals the forwarded amount, which can be zero. The position NFT contract does not validate liquidity on initialization, so a zero-liquidity NFT is created. This behavior does not directly compromise funds but can lead to on-chain clutter, inconsistent UX expectations, and additional indexing or storage overhead.

```
fun depositNFT(senderAddress: address, queryId: int, inMsgBody : slice,
nftContainer : PositionNFTStorage) {
    var liquidity2Deposit: int = inMsgBody.loadUint(128);
    ...
```

Figure 9.1: `depositNFT` accepts the deposit amount with no nonzero validation. (contracts/positionnft_tolk/operations/positionnft_burn.tolk#L101–L102)

```
body: DepositMessage {
    query_id  : queryId,
    index     : nftContainer.index,
    lpProvider  : nftContainer.userAddress,
    positions_cell : ({
        pos0 : ({
            index    : nftContainer.index,
            subindex : 0,
            data     : position0ToDeposit,
            liquidity2Burn : liquidity2Deposit
        } as BurnOrder).toCell()
    } as BurnStorage).toCell(),
```

Figure 9.2: Deposit forwards the amount as `liquidity2Burn` in the burn-order structure. (contracts/positionnft_tolk/operations/positionnft_burn.tolk#L136–L147)

Recommendations

Short term, reject zero-liquidity deposits and do not mint positions with zero liquidity in the deposit/reforge flow; treat zero as a no-op or fee-only collection path.

Long term, add tests checking that deposit/reforge never produces zero-liquidity NFTs, and that zero-liquidity inputs only collect fees and do not mutate position state or counters.

10. Missing zero address check in the reforgeOperation function can lead to loss of funds

Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-TONCO-10
Target: `contracts/pool/operations/pool_reforge.func`	

Description

The pool contract's `reforgeOperation` function is missing a zero address check when overriding the `pay_to_target` variable to the ALM address. If this address is zero, all jettons and positions supplied with the reforge message will be permanently lost.

The `reforgeOperation` function allows the `pay_to_target` variable to be overwritten if the user supplies 1 as the `action_target` in the payload, which it later uses as the receiver of the `ROUTER_OPERATION_PAY_TO` message.

```
if ((source == REFORGE_SOURCE_USER) | (source == REFORGE_SOURCE_ALM))
{
    if (action_target == 1) {
        pay_to_target = pool::roles.at(POOL_ROLE_ALM);
    }
}
```

Figure 10.1: A snippet of the `reforgeOperation` function (`contracts/pool/operations/pool_reforge.func#L192–L197`)

However, if this address is zero, the message will immediately fail and the jettons and positions used will be permanently lost.

Exploit Scenario

Alice mistakenly sets the `action_target` flag to 1 when depositing liquidity even though the ALM address is not set in the pool. She loses her jettons and the liquidity position.

Recommendations

Short term, add a check to the code in figure 10.1 that ensures the ALM address is set to a valid address.

Long term, improve the test suite by testing for common adversarial and misconfiguration scenarios.

11. Multihop shortcut allows a position to be minted through the router contract's account

Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-TONCO-11
Target: `contracts/router_tolk/operations/router_send.tolk`	

Description

If the `POOL_OPERATION_FUND_ACCOUNT` operation is attached to a multihop cell during a multihop shortcut, the liquidity will be deposited by using the router contract's account.

The multihop shortcut path immediately routes the attached multihop payload to the `transferNotificationPayloadProcess` function, which allows three different operations to be performed: a swap, a liquidity deposit on behalf of another user, and a liquidity deposit on behalf of the sender. During a multihop shortcut, the sender is set to the router contract's address:

```
if ((router.flags & ROUTER_FLAG_MULTIHOP_SHORTCUT) != 0) {
    if (coins.amount0 > 0) {
        if (payToBody.receiver0 == contract.getAddress()) {
            transferNotificationPayloadProcess(queryId, coins.amount0,
contract.getAddress(), coins.jetton0Address, payloadCell0, tonAmount0);
            coins.amount0 = 0;
        }

        if (payToBody.receiver0 == router.proxyTonWallet) {
            transferNotificationPayloadProcess(queryId, coins.amount0,
contract.getAddress(), router.proxyTonWallet, payloadCell0, tonAmount0);
            coins.amount0 = 0;
        }
    }
 // ...
```

Figure 11.1: A snippet of the multihop shortcut logic in the router contract (`contracts/router_tolk/operations/router_send.tolk#L98–L127`)

The `POOL_OPERATION_FUND_ACCOUNT` operation is intended to be used by users to deposit liquidity through their own accounts; however, in the multihop shortcut branch, the liquidity will be deposited through the router's account instead. If a user mistakenly uses this operation, a malicious user can potentially introduce a race condition in order to steal their deposited jettons from the router's account by overriding the order attached with the payload.

Recommendations

Short term, add a check that ensures the `POOL_OPERATION_FUND_ACCOUNT` cannot be used in a multihop.

Long term, improve the test suite by adding multihop tests for all the allowed operations.

12. Malformed multihop cell can lead to jetton loss

Severity: **Medium**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-TONCO-12
Target: `contracts/router_tolk/operations/router_receive.tolk`	

Description

Attaching a malformed `multihopCell` to a multihop swap can result in a cell underflow panic, leading to the loss of all the jettons supplied to this operation.

When the router contract processes a swap operation, it checks for an optional `multihopCell` attached to the message data. If the `multihopCell` is null, the router will build a default cell on behalf of the user:

```
var multihopCell: cell? = command.loadMaybeRef();

/* Payload is zeroed at entry point */
if ((multihopCell != null) & ((router.flags & ROUTER_FLAG_PAYLOADS) != 0)) {
} else {
    multihopCell = beginCell()
        .storeAddress(ownerAddress)
        // .storeCoins(0)
        // .storeMaybeRef()
        // .storeCoins(0)
        // .storeMaybeRef()
        // .storeAddress(createAddressNone())
        .storeZeroes(4+1+4+1+2)
    .endCell();
}
```

Figure 12.1: A snippet of the `transferNotificationPayloadProcess` function (contracts/router_tolk/operations/router_receive.tolk#L53–L66)

However, if the `multihopCell` is not null, the cell is never checked to have the correct format. Therefore, a malformed cell can be attached to the message data and passed along to the pool contract's `POOL_OPERATION_SWAP` operation, which will attempt to parse it:

```
cell  action_cell          = in_msg_body~load_ref();
slice action_slice         = action_cell.begin_parse();
    slice target_address       = action_slice~load_msg_addr();
    int   ok_forward_amount    = action_slice~load_coins();
    cell  ok_forward_payload   = action_slice~load_maybe_ref();
    int   ret_forward_amount   = action_slice~load_coins();
    cell  ret_forward_payload  = action_slice~load_maybe_ref();
```

```
    slice excess_address       = action_slice~load_msg_addr();
```

Figure 12.2: A snippet of the `swapOperation` function (`contracts/pool/operations/pool_swap.func#L29–L36`)

If the cell does not contain enough data, this will cause a cell underflow panic, causing the swap to fail and the jettons supplied to the swap to be permanently lost.

Exploit Scenario

Alice mistakenly builds a malformed `multihopCell` by forgetting to add the `excess_address` to the cell data. She initiates a swap that includes this payload; however, the swap fails as soon as the jetton is received by the router. Alice permanently loses the jettons she has transferred.

Recommendations

Short term, add a check of the remaining bits and refs of the `multihopCell` in order to ensure it follows the correct data format. If it does not, have the router overwrite it with the default cell or ignore the multihop and refund the jettons to the user.

Long term, improve the testing suite by adding tests for scenarios where malformed data is sent with the message.

13. Router will keep the jettons if the transfer notification contains an unsupported operation

Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-TONCO-13
Target: `contracts/router_tolk/operations/router_receive.tolk`	

Description

If the transfer notification payload contains an operation that is not one of the three supported operations, the jettons that the user transferred will be permanently lost.

The `transferNotificationPayloadProcess` function is triggered when a jetton is transferred to the router with a forward payload. The forward payload includes the operation that should be executed by the router. However, if the operation is anything other than the three predefined operations, the router will simply keep the jettons.

```
fun transferNotificationPayloadProcess(queryId: int, jettonAmount: int, fromUser:
address, senderAddress: address, commandBody: cell?, tonLimit: int)
{
    if (commandBody == null) {
        return;
    }

    var command: slice = commandBody.beginParse();

    if (command.isEndOfBits()) {
        return;
    }

    var transferredOp: int     = command.loadUint(32);
    var jettonWallet1: address = command.loadAddress();

    var router : RouterStorage = lazy RouterStorage.fromCell(contract.getData());

    if (transferredOp == POOL_OPERATION_SWAP) {
        // ...
    }

    if (transferredOp == POOL_OPERATION_FUND_ACCOUNT) {
        // ...
    }

    if (transferredOp == POOL_OPERATION_FUND_SOMEONES_ACCOUNT) {
        // ...
```

```
    }
}
```

Figure 13.1: A snippet of the `transferNotificationPayloadProcess` function (`contracts/router_tolk/operations/router_receive.tolk#L17–L208`)

Exploit Scenario

Alice transfers her jettons to the router and attaches a forward payload with an incorrect operation. The message execution succeeds and the router silently keeps the jettons.

Recommendations

Short term, have the router refund the jettons when an invalid operation is attached to the message body.

Long term, retroactively write a system specification that details all the failure cases and ensures the major failure cases can refund user jettons or positions.

14. Pool reinitialization risks	
Severity: **Low**	Difficulty: **High**
Type: Configuration	Finding ID: TOB-TONCO-14
Target: `contracts/pool/operations/pool_create.func`	

Description

The pool and router admin could misconfigure important pool parameters by reinitializing the pool.

The admin of a pool contract can reinitialize the contract by sending a message with the `POOL_OPERATION_INIT` operation to the pool. The admin of the router contract can do the same operation through the router. Both of them are able to update the following configuration parameters:

- The privileged roles in the system, including the `POOL_ROLE_ADMIN`, `POOL_ROLE_CONTROLLER`, `POOL_ROLE_CREATOR`, and `POOL_ROLE_ORACLE` roles
- The tick spacing used in the pool
- The lock status of the pool
- The protocol fee, base fee, and current fee
- The current square root price and the current tick, if no liquidity positions have been minted in the given pool

While these roles are considered trusted, there exist ways in which the pool could be misconfigured by this action:

- Setting a different tick spacing on a pool that holds any amount of liquidity can cause the pool to malfunction and prevent users from interacting with their positions due to the tick validity checks.
- Setting the fee percentage amounts to 100% will result in a pool that is practically inoperable.
- Setting the `POOL_ROLE_ADMIN` role in a single step is error-prone; if an incorrect address is set, the router administrator is the only one that can correct this mistake.

Recommendations

Short term, implement the following changes:

- Do not allow the tick spacing to change if liquidity positions have already been minted for the given pool.
- Add a reasonable upper bound to the fee percentage configuration parameters.
- Move the `POOL_ROLE_ADMIN` update to the two-step `POOL_OPERATION_CHANGE_ADMIN_START` and `POOL_OPERATION_CHANGE_ADMIN_COMMIT` operations.

Long term, test all the administrator actions with various pool configurations, including updating configurations on a pool with and without liquidity.

15. Return value of modifyPosition is ignored inside burnPosition

Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-TONCO-15
Target: `contracts/pool/operations/pool_burn.func`	

Description

The `burnPosition` function does not check the return value of `modifyPosition`, which can return error codes to indicate failures during position modification (figure 15.1). In contrast, the `mintPosition` function properly validates the return value of `modifyPosition` and throws an exception if an error is returned (figure 15.2). This inconsistency represents a deviation from the pattern established elsewhere in the codebase.

```
(int amount0, int amount1) =  computeAmounts(tickLower, tickUpper, liquidityDelta);
modifyPosition(tickLower,tickUpper, liquidityDelta);
```

Figure 15.1: Missing return value check in `burnPosition` (`contracts/pool/operations/pool_burn.func#L178–L179`)

The `modifyPosition` function is designed to return error codes in several scenarios, including when there are too many ticks (`RESULT_TOO_MANY_TICKS`) or when there is too much liquidity per tick (`RESULT_TOO_MUCH_LIQUIDITY`). While currently only the mint scenario can trigger these early returns, making them unreachable during burn operations, the function signature includes an explicit return value that should be checked for consistency and defensive programming practices.

```
rejectMint = modifyPosition(tickLower, tickUpper, liquidity);
dumpint("   modifyPosition returned rejectMint:" , rejectMint);
if (rejectMint != 0) {
    throw (rejectMint);
}
```

Figure 15.2: Return value check in `mintPosition` (`contracts/pool/operations/pool_mint.func#L145–L149`)

Recommendations

Short term, add a return value check in `burnPosition` immediately after calling `modifyPosition`, similar to the implementation in `mintPosition`. `burnPosition` should store the return value and throw an exception if it is nonzero. This will ensure consistency

across the codebase and provide defensive programming in case future modifications to `modifyPosition` introduce additional error conditions that could affect burn operations.

Long term, review all calls to functions that return error codes throughout the codebase and ensure that return values are consistently checked. Consider implementing a coding standard or linting rule that enforces checking the return values of functions that can return error codes. This will prevent inconsistencies from being introduced and allow the development team to maintain a uniform error-handling pattern across the codebase.

16. Router does not check for pool existence

Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-TONCO-16
Target: `contracts/router_tolk/operations/router_receive.tolk`	

Description

The `transferNotificationPayloadProcess` function does not verify whether a pool contract has been deployed before forwarding user funds to it. When a user sends `POOL_OPERATION_FUND_ACCOUNT` or `POOL_OPERATION_FUND_SOMEONES_ACCOUNT` operations with jetton tokens for a pool that has not been created, the router calculates the pool address and sends a non-bounceable message to it. In TON, non-bounceable messages sent to nonexistent contracts do not bounce back to the sender but instead credit the funds to that address. This represents a user error scenario where funds could be locked if the pool is never deployed.

```
    var fundMessage = createMessage({
         bounce : false,
         value : tonLimit,
         dest : poolAddress,
         body : msgBody.endCell()
     });

     fundMessage.send(tonLimit == 0 ? SEND_MODE_CARRY_ALL_REMAINING_MESSAGE_VALUE :
SEND_MODE_REGULAR);
     Return;
}
```

Figure 16.1: Non-bounceable message sent to potentially undeployed pool (contracts/router_tolk/operations/router_receive.tolk#L158–L167)

Recommendations

Short term, change the funding messages to use `bounce: true` and implement bounce handling in the router. In TON, messages sent to nonexistent contracts always bounce back to the sender when the bounce flag is set. The router can then handle the bounced message and return the jettons to the user. Alternatively, clearly document this risk in user-facing documentation and interfaces to warn users that sending fund account operations to nonexistent pools will result in irreversible loss of funds.

Long term, consider implementing pool existence validation in the router contract that runs before fund account operations are forwarded. This could be achieved through a

registry of deployed pools, and this will prevent accidental fund loss from user errors when specifying jetton pairs.

17. Lack of transparency in timelocked code updates

Severity: **Informational**	Difficulty: **Low**
Type: Auditing and Logging	Finding ID: TOB-TONCO-17
Target: `contracts/router_tolk/operations/router_update.tolk`, `contracts/pool/operations/pool_update.func`	

Description

Both the router and pool contracts implement timelock mechanisms for code updates, allowing admins to schedule code changes that take effect after a delay period. This design enables users to review pending changes and take action if they disagree with updates. However, neither contract provides a getter method to retrieve the pending code or its hash, making it impossible for users to verify what code will be deployed when the timelock expires.

```
if (timelock.newCode != null) {
    contract.setCodePostponed(timelock.newCode);
}
```

Figure 17.1: Code timelock set without transparency mechanism (contracts/router_tolk/operations/router_update.tolk#L168–L170)

Recommendations

Short term, add getter methods that return the hash of the pending code update, along with the timelock expiration time, to both the router and pool contracts. This will allow users to verify off-chain what code changes are scheduled.

Long term, enhance transparency by providing comprehensive getter methods that expose all pending timelocked changes, including admin address changes, proxy TON wallet updates, and flag modifications. This will create a trustless environment where users can verify all pending changes before they take effect.

18. Manual balance calculation instead of raw_reserve

Severity: **Informational**	Difficulty: **Low**
Type: Configuration	Finding ID: TOB-TONCO-18
Target: `contracts/router_tolk/operations/router_gas.tolk`, `contracts/positionnft_tolk/operations/positionnft_transfer.tolk`, `contracts/account_tolk/account.tolk`	

Description

Several contracts manually calculate the amount to send by subtracting a reserve amount from the current balance, rather than using the `raw_reserve` function followed by sending the message with `value: 0` and `SEND_MODE_CARRY_ALL_REMAINING_MESSAGE_VALUE`. This manual approach is more error-prone and does not properly account for forward fees, which can lead to edge cases causing the remaining balance to be lower than expected after message delivery.

```
assert(myBalance > ROUTER_TON_RESERVE) throw INSUFFICIENT_GAS;

//sendEmptyMessage(myBalance - ROUTER_TON_RESERVE, `router::admin_address`,
MODE_NORMAL);
 val resetGasMessage = createMessage({
    bounce: false,
    value : myBalance - ROUTER_TON_RESERVE,
    dest  : router.adminAddress
});
resetGasMessage.send(SEND_MODE_REGULAR);
```

Figure 18.1: Manual balance calculation in router gas reset (contracts/router_tolk/operations/router_gas.tolk#L10–L18)

```
var restAmount = contract.getOriginalBalance() - POSITION_NFT_MIN_STORAGE_TONS;
if (msg.forwardTonAmount) {
    restAmount -= (msg.forwardTonAmount + fwdFee);
}
var needResponse = msg.sendExcessesTo.isInternal();
if (needResponse) {
    assert (msg.sendExcessesTo.getWorkchain() == BASECHAIN) throw WRONG_WORKCHAIN;
    restAmount -= fwdFee;
}
```

Figure 18.2: Manual balance calculation in NFT transfer (contracts/positionnft_tolk/operations/positionnft_transfer.tolk#L45–L53)

```
if (op == ACCOUNT_OPERATION_RESET_GAS) {
    assert(myBalance > REQUIRED_TON_RESERVE) throw INSUFFICIENT_GAS;

    val resetGasMessage = createMessage({
        bounce: true,
        value: myBalance - REQUIRED_TON_RESERVE,
        dest: account.userAddress
    });
    resetGasMessage.send(SEND_MODE_REGULAR);
    Return;
}
```

Figure 18.3: Manual balance calculation in account gas reset (contracts/account_tolk/account.tolk#L72–L82)

Recommendations

Short term, replace manual balance calculations with the `raw_reserve` pattern in the identified functions. Use `raw_reserve(RESERVE_AMOUNT, RESERVE_REGULAR)` to reserve the minimum required balance and have the code send messages with `value: 0` and `SEND_MODE_CARRY_ALL_BALANCE`. This will ensure that all remaining balance after the reservation is sent, with fees automatically deducted from the message value rather than the contract balance.

Long term, review all future contracts for similar patterns and apply the `raw_reserve` approach consistently throughout the codebase. This will ensure uniform balance management across all contracts and reduce the likelihood that similar issues will be introduced in future development.

19. Users are able to route positions to the ALM

Severity: **Undetermined**	Difficulty: **Low**
Type: Access Controls	Finding ID: TOB-TONCO-19
Target: `contracts/account_tolk/operations/order_manage.tolk`	

Description

Users have full control over the order structure when calling `parseOrder`, including the `target_action` field that determines whether newly created positions should be deposited into the ALM account. The `target_action` field is directly set by the user and used in the reforge operation to route positions to the ALM. According to the TONCO team, unprivileged users should not have control over positions sent to the ALM, yet the current implementation allows any user to set `target_action` to 1 in their order and route positions to the ALM account.

```
if ((source == REFORGE_SOURCE_USER) | (source == REFORGE_SOURCE_ALM))
{
    if (action_target == 1) {
        pay_to_target = pool::roles.at(POOL_ROLE_ALM);
    }
}
```

Figure 19.1: Action target determines ALM routing.
(contracts/pool/operations/pool_reforge.func#L192–L197)

The ALM component was out of scope for this audit, so the exact impact of unauthorized position deposits depends on how the ALM handles incoming positions. If the ALM account has the same four-position limitation as normal accounts, an attacker could intentionally race the ALM to fill all available slots with malicious or low-value positions. This would cause legitimate valuable positions intended for the ALM to be rejected or lost, potentially resulting in loss of funds. Given that the ALM is presumed to hold substantial funds for automated liquidity management, the impact of losing valuable positions could be significant.

Recommendations

Short term, remove `(source == REFORGE_SOURCE_USER)` from the conditions that allow positions to be routed to the ALM account in the reforge operation. Only `REFORGE_SOURCE_ALM` should be permitted to route positions to the ALM account, ensuring the ALM remains a closed subsystem accessible only to authorized sources.

Long term, expand the test suite to include access control tests that verify unprivileged users cannot trigger privileged operations. Specifically, add test cases that attempt to send positions from normal user accounts and verify that positions are not routed to the ALM. Include tests for all privileged subsystems to ensure proper access control enforcement across the codebase.

20. Excess TON refunded to the wrong address in swaps

Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-TONCO-20
Target: `contracts/pool/operations/pool_swap.func`	

Description

In swap operations, the pool refunds jettons and excess TON to recipients based on swap direction. When `zeroForOne` is false, the pool sets `receiver0` to `target_address` rather than `owner_address`, and the router uses `receiver0` as the default excess TON receiver for both jetton transfers. This means that when `zeroForOne` is false, excess TON from both jetton transfers is sent to `target_address` instead of the swap originator's `owner_address`.

```
var excess_receiver: address  = payToBody.receiver0;
```

Figure 20.1: Router defaults excess receiver to `receiver0`.
(`contracts/router_tolk/operations/router_send.tolk#L46`)

Recommendations

Short term, modify the pool to always set a dedicated excess receiver field to `owner_address` in the payload, or ensure that `excess_address` from the swap operation parameters is properly propagated to the router as the excess receiver. This will ensure that excess TON refunds are sent to the swap originator regardless of swap direction or target address configuration.

Long term, enhance the test suite to include test cases that verify excess TON is refunded to the correct address for both swap directions. This will prevent similar address confusion issues from being introduced in future code changes.

21. The router's TON balance can be drained via negative amount calculation

Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-TONCO-21
Target: `contracts/router_tolk/operations/router_send.tolk`	

Description

The TON balance of the router can be drained if the multihop shortcut path is enabled due to a negative amount computation.

The router's `payToOperation` function calculates TON amounts to attach to jetton transfer messages by subtracting upkeep costs and payload amounts from the incoming message value. This calculation can result in a negative `tonAmount` value, which is then divided between two transfers. Even when `tonAmount` is negative, one of the resulting values (`tonAmount0` or `tonAmount1`) can be positive while the other is negative. An attacker can exploit this by using the multihop shortcut feature with an invalid operation on the transfer that would have a negative TON amount, preventing serialization errors while causing the router to spend TON from its own balance on the positive-amount transfer.

```
// Either one or both amounts must be non-zero
if ((coins.amount0 > 0) & (coins.amount1 > 0)) {
    // Divide remaining ton_amount between two transactions
    var tonAmount: int = (msgValue - (ROUTER_TON_UPKEEP + ROUTER_PAY_TO_GAS)) -
payloadAmount0 - payloadAmount1;
    tonAmount0 = payloadAmount0 + tonAmount / 2;
    tonAmount1 = payloadAmount1 + tonAmount / 2;
    mode = SEND_MODE_REGULAR;
```

Figure 21.1: TON amount calculation can produce negative values. (contracts/router_tolk/operations/router_send.tolk#L81–L87)

An example calculation is presented in figure 21.2:

```
msgValue = 0
ROUTER_TON_UPKEEP + ROUTER_PAY_TO_GAS = 12,417,000
payloadAmount0 = 200,000,000
payloadAmount1 = 100,000,000

tonAmount = 0 - 12,417,000 - 200,000,000 - 100,000,000
tonAmount = - 312,417,000

tonAmount0 = 200,000,000 + (-312,417,000 / 2) = 43,791,500
tonAmount1 = 100,000,000 + (-312,417,000 / 2) = - 56,208,500
```

Figure 21.2: Example `tonAmount` calculation

Normally, converting a negative value to `coins` would cause a serialization error. However, if the multihop shortcut is enabled via the `ROUTER_FLAG_MULTIHOP_SHORTCUT` flag and the attacker provides an invalid operation code in the payload for `jetton1`, the `transferNotificationPayloadProcess` function becomes a no-op and returns early without processing the payload (as shown in TOB-TONCO-13).

This prevents the negative `tonAmount1` from being serialized, preventing the error. The router then sends a message with a `tonAmount0` value of `43,791,500` using TON from its own balance, allowing the attacker to drain router funds.

```
if (coins.amount1 > 0) {
    if (payToBody.receiver1 == contract.getAddress()) {
        dumpstr("    Shortcut to router with coins 1");
        transferNotificationPayloadProcess(queryId, coins.amount1,
contract.getAddress(), coins.jetton1Address, payloadCell1, tonAmount1);
        coins.amount1 = 0;
    }
```

Figure 21.3: Multihop shortcut bypasses serialization. (contracts/router_tolk/operations/router_send.tolk#L114–L119)

Exploit Scenario

Eve initiates a swap operation by sending jettons to the router with a crafted `multihopCell` containing high `payloadAmount0` and `payloadAmount1` values (e.g., 200,000,000 and 100,000,000 nanotons). She sets `receiver1` to the router address to trigger the multihop shortcut and includes an invalid operation code in `payloadCell1`. The router forwards this swap to the pool, which processes it and sends a `ROUTER_OPERATION_PAY_TO` message back to the router with `value: 0` and `SEND_MODE_CARRY_ALL_REMAINING_MESSAGE_VALUE`, resulting in a very small or zero-value `msgValue`. The router's `payToOperation` calculates `tonAmount = 0 - 12,417,000 - 200,000,000 - 100,000,000 = -312,417,000`, resulting in a `tonAmount0` value of `43,791,500` (positive) and a `tonAmount1` of `-56,208,500` (negative). The multihop shortcut processes `jetton1` by calling `transferNotificationPayloadProcess`, which encounters the invalid operation and returns early without serializing the negative `tonAmount1`. However, the router still sends the `jetton0` transfer with a `tonAmount0` value of `43,791,500` using TON from its own balance. By repeatedly exploiting this vulnerability, or by using a very large value for one of the payload amounts, Eve drains the router's TON balance.

Recommendations

Short term, add validation to ensure that `tonAmount0` and `tonAmount1` are both nonnegative before proceeding with message sending. If either value is negative, the code should reject the operation and return an error. This will prevent the router from spending its own TON on behalf of users who have not provided sufficient funds.

Long term, enhance the test suite to include comprehensive tests for edge cases in TON amount calculations, including scenarios with zero or minimal `msgValue`, high payload amounts, and multihop shortcuts.

22. Overflow in getMaxLiquidityForAmount0Precise can result in fund loss

Severity: **Medium**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-TONCO-22
Target: `contracts/pool/amounts.func`	

Description

The `getMaxLiquidityForAmount0Precise` function can overflow because multiplication is performed before division, potentially leading to a loss of jettons when a user attempts to mint liquidity through the maximum liquidity path.

The `computeLiquidity` function is used to calculate the maximum amount of liquidity that a user can mint using the jetton amounts they transferred; specifically, the function is used when a user passes the `MINT_AS_MUCH_AS_POSSIBLE` operation to the `mintPosition` function. The `computeLiquidity` function calls the `getMaxLiquidityForAmounts` function to perform the calculation. If the current square root price is within or below the tick bounds of the position, `getMaxLiquidityForAmounts` will call `getMaxLiquidityForAmount0Precise`:

```
int getMaxLiquidityForAmount0Precise(int sqrtRatioAX96, int sqrtRatioBX96, int
amount0)
;; method_id
{
    if (sqrtRatioAX96 > sqrtRatioBX96) {
        [sqrtRatioAX96, sqrtRatioBX96] = [sqrtRatioBX96, sqrtRatioAX96];
    }
    int numerator = (amount0 * sqrtRatioAX96) * sqrtRatioBX96;
    int denominator =  Q96 * (sqrtRatioBX96 - sqrtRatioAX96);
    return numerator / denominator;
}
```

Figure 22.1: The overflow in `getMaxLiquidityForAmount0Precise` (`contracts/pool/amounts.func#L31–L40`)

However, on the highlighted line of figure 22.1, we can see that three potentially large values are multiplied at once. This calculation can easily overflow the 257-bit signed integer when the amount or prices are large, causing the message to fail and the jettons to be lost.

Exploit Scenario

Alice attempts to deposit jettons into a pool between two large ticks using the maximum liquidity path. Her transaction fails due to an overflow in the `getMaxLiquidityForAmount0Precise` function and she permanently loses her jettons.

Recommendations

Short term, use the same approach used in Uniswap v3: calculate an intermediate value first before the final value in order to avoid overflows in the multiplication step. Use `muldiv` to take advantage of the 513 bit precision.

Long term, use stateless fuzzing to test all of the calculations performed in the protocol for overflow risks.

23. Incomplete handling of swap exceptions

Severity: **Informational**	Difficulty: **Low**
Type: Error Reporting	Finding ID: TOB-TONCO-23
Target: `contracts/pool/operations/pool_swap.func`	

Description

The swap operation's exception handling checks only for the `LOWER_THEN_MIN_OUT` exception and sets the exit code accordingly. If any other exception occurs during the swap execution, such as `POOL_INACTIVE` or exceptions from `swapInternal`, the exit code remains set to `RESULT_SWAP_OK` even though the swap failed. This results in incorrect status reporting where failed swaps are marked as successful, which will cause issues for indexers and off-chain systems that rely on the exit code to determine swap outcomes.

```
int exitCode = RESULT_SWAP_OK;
{-
    Try/Catch would revert to this point in case of exception.
    At this point reserves are changed already we can't call load_storage(). Also it
would be too gas heavy
    -}
try {
    int overrideLock = equal_slice_bits(owner_address,
pool::roles.at(POOL_ROLE_CONTROLLER)) |
                       equal_slice_bits(owner_address,
pool::roles.at(POOL_ROLE_CREATOR));

    if (((pool::flags & FLAG_LOCK_MASK) != 0) & (~ overrideLock))  {
        throw (POOL_INACTIVE);
    }
   ;; ... swap logic ...
} catch (_, int exceptionId) {
    if (exceptionId == LOWER_THEN_MIN_OUT) {
        dumpstr("   Reverting because of the min out");
        exitCode = RESULT_SWAP_OUTPUT_TOO_SMALL;
    }
```

Figure 23.1: Incomplete exception handling in swap operation (`contracts/pool/operations/pool_swap.func#L83–L134`)

Recommendations

Short term, add an `else` clause to the exception handler that sets a generic failure exit code for any exception other than `LOWER_THEN_MIN_OUT`. This will ensure that failed swaps are never reported as successful, allowing indexers and monitoring systems to correctly identify swap failures.

Long term, implement comprehensive exception handling that maps each exception type to a specific exit code. Add test cases that trigger different exception scenarios and verify that the appropriate exit codes are returned. This will improve observability and help off-chain systems diagnose swap failures.

24. Depositing more than four positions causes failure

Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-TONCO-24
Target: `contracts/account_tolk/operations/account_refill.tolk`	

Description

Account contracts are limited to holding a maximum of four positions. When a user attempts to deposit positions that would exceed this limit, the operation fails with a `POSITION_OVERFLOW` error. This causes the entire transaction to revert, and the positions intended for deposit are lost. While the TONCO developers seem to be aware of this limitation, based on code comments, the issue exists and can lead to unexpected failures and loss of liquidity for users who are unaware of the constraint.

```
/*
    This is actually quite a big problem. Obviously UI could control all the
communications and disallows
    depositing more then 4 NFTs, because only the user himself can do it.
        However this really makes design fragile, alternative could be - to
passthrough with POOL_REFORGE
    For this we have to reserve gas.
    Also even with this exception it would still be nice to store coins parsing.
Should try/catch this
*/
assert (inCount + accCount <= 4) throw POSITION_OVERFLOW;
```

Figure 24.1: Four-position limit enforcement (contracts/account_tolk/operations/account_refill.tolk#L75–L83)

Recommendations

Short term, implement graceful handling when the position limit is exceeded. Instead of reverting the entire transaction, have the code accept the coins but reject the excess positions; alternatively, implement a try-catch block to preserve the coin deposits while excess positions are being returned to the user. Document this limitation clearly in user-facing documentation and interfaces.

Long term, enhance the test suite to include test cases that verify proper handling of position overflow scenarios. Test cases should cover scenarios where users attempt to deposit positions that would exceed the four-position limit and verify that funds are not lost.

25. Incorrect price caching corrupts legacy tick format

Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-TONCO-25
Target: `contracts/pool/amm.func`	

Description

The price caching mechanism in `swapInternal` incorrectly processes ticks stored in the legacy format, corrupting their data structure and causing subsequent operations to fail and pool liquidity to get highly inflated. When a legacy tick with `liquidityTotal` <= 2^96 - 1 is encountered, the first 160 bits of its 256-bit `liquidityTotal` field are zero. The code misinterprets this as an uncached price, drops these 160 bits, and prepends a calculated `sqrtPriceNext` value. This transforms the 896-bit legacy tick structure into a corrupted format in which `sqrtPriceNext` is part of `liquidityTotal`, resulting in an incorrectly inflated liquidity value that triggers denial of service in mint and burn operations.

```
if (sqrtPriceNext == 0) { ;; For legacy tick: first 160 bits are 0 (part of
liquidityTotal)
    sqrtPriceNext = getSqrtRatioAtTick(tickNextNumber);
    if (hasEntry != 0) {
        tickNext~load_uint(160); ;; Drops first 160 bits (zeros from liquidityTotal)
        slice tickWithCache = begin_cell()
            .store_uint(sqrtPriceNext, 160) ;; Prepends calculated price
            .store_slice(tickNext) ;; Appends remaining tick data
        .end_cell().begin_parse();
        tickNext = tickWithCache;
    }
} else {
    ;;~strdump("Cache loaded");
}
```

Figure 25.1: Price caching logic corrupts legacy tick format. (`contracts/pool/amm.func#L328–L340`)

The legacy tick format stores `[liquidityTotal 256 | liquidityDelta 128 | outerFeeGrowth0Token 256 | outerFeeGrowth1Token 256]` in 896 bits. After the incorrect caching operation, it becomes `[sqrtPriceNext 160 | tail of liquidityTotal 96 | liquidityDelta 128 | outerFeeGrowth0Token 256 | outerFeeGrowth1Token 256]`, still 896 bits. When `load_tick` processes this corrupted tick, it incorrectly interprets the data:

```
(int, int, int, int, int) load_tick(slice tickSlice) impure inline {
    if (slice_bits(tickSlice) == 896) {
        int liquidityTotal       = tickSlice~load_uint(256);
        int liquidityDelta       = tickSlice~load_int (128);
        int outerFeeGrowth0Token = tickSlice~load_uint(256);
        int outerFeeGrowth1Token = tickSlice~load_uint(256);
        return (liquidityTotal, liquidityDelta, outerFeeGrowth0Token,
outerFeeGrowth1Token, 0);
    }
    ;; ... new format handling
}
```

Figure 25.2: `load_tick` misinterprets corrupted legacy tick. (`contracts/pool/pool_storage.func#L366–L384`)

The corrupted tick still has 896 bits, so it matches the legacy format check. However, `liquidityTotal` is now read as the first 256 bits, which includes the prepended `sqrtPriceNext` (160 bits) concatenated with the remaining 96 bits of the original `liquidityTotal`. This results in a massively inflated liquidity value that exceeds `maxTickLiquidity` and causes `modifyPosition` to return `RESULT_TOO_MUCH_LIQUIDITY`.

Recommendations

Short term, add validation to detect legacy format ticks before applying price caching. Check if the tick has 896 bits total length and skip the caching operation for legacy format ticks. Alternatively, since the legacy format is no longer in use according to the development team, remove support for it entirely to eliminate this code path.

Long term, implement versioning for tick data structures to handle format migrations cleanly. Include explicit format identifiers or version numbers in the data structure to prevent misinterpretation of different formats. Document the expected tick formats and their evolution to prevent similar issues when future format changes are needed.

26. Rounding difference between Uniswap v3 and TONCO amount delta calculations

Severity: **Undetermined**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-TONCO-26
Target: `contracts/pool/liquidity.func`	

Description

The `getAmount0Delta` and `getAmount1Delta` functions return a different result than the equivalent functions in the Uniswap v3 implementation. The TONCO functions have a difference of 1 unit, favoring the user when minting or burning liquidity. While we did not find a way to exploit this discrepancy, we recommend further investigating this issue and fully triaging its root cause.

Recommendations

Short term, use the differential fuzzing suite to detect and triage the issue.

Long term, enhance the differential fuzzing suite to fuzz all of the math that should be equivalent to the Uniswap v3 implementation. Improve the fuzzing suite by adding stateful tests.

27. Potential overflow in fee growth computation

Severity: **Undetermined**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-TONCO-27
Target: `contracts/pool/ticks.func`	

Description

The TONCO implementation for computing fee growth inside the position uses signed 256-bit integers for computation where arithmetic overflow and underflow will cause an error to be thrown and the message execution to fail. This represents a notable difference between this implementation and the Uniswap v3 implementation, which intentionally allows for wrapping overflows and underflows in order to compute correct values when collecting liquidity provider fees.

While our testing efforts did not discover a specific case where the overflow or negative growth calculations enable the protocol to be exploited, additional effort should be made to thoroughly test this difference.

```
(int, int) computeFeeGrowthInside(
    int tickLower,
    int tickUpper,
    int tickCurrent,
    int feeGrowthGlobal0X128,
    int feeGrowthGlobal1X128
)
method_id
{

    (_, _, int lowOuterFeeGrowth0Token, int lowOuterFeeGrowth1Token, _, _) =
load_tick_default(tickLower);
    (_, _, int  upOuterFeeGrowth0Token, int  upOuterFeeGrowth1Token, _, _) =
load_tick_default(tickUpper);

    ;; calculate fee growth below
    int feeGrowthBelow0X128 = 0;
    int feeGrowthBelow1X128 = 0;
    if (tickCurrent >= tickLower) {
        feeGrowthBelow0X128 = lowOuterFeeGrowth0Token;
        feeGrowthBelow1X128 = lowOuterFeeGrowth1Token;
    } else {
        feeGrowthBelow0X128 = feeGrowthGlobal0X128 - lowOuterFeeGrowth0Token;
        feeGrowthBelow1X128 = feeGrowthGlobal1X128 - lowOuterFeeGrowth1Token;
    }
```

```
    ;; calculate fee growth above
    int feeGrowthAbove0X128 = 0;
    int feeGrowthAbove1X128 = 0;
    if (tickCurrent < tickUpper) {
        feeGrowthAbove0X128 = upOuterFeeGrowth0Token;
        feeGrowthAbove1X128 = upOuterFeeGrowth1Token;
    } else {
        feeGrowthAbove0X128 = feeGrowthGlobal0X128 - upOuterFeeGrowth0Token;
        feeGrowthAbove1X128 = feeGrowthGlobal1X128 - upOuterFeeGrowth1Token;
    }

    return (
        feeGrowthGlobal0X128 - feeGrowthBelow0X128 - feeGrowthAbove0X128,
        feeGrowthGlobal1X128 - feeGrowthBelow1X128 - feeGrowthAbove1X128
    );

}
```

Figure 27.1: The `computeFeeGrowthInside` function (`contracts/pool/ticks.func#L137–L195`)

Recommendations

Short term, enhance and run the differential fuzzing test suite in order to determine if the overflow is achievable in a production environment.

Long term, enhance the differential fuzzing suite to include stateful fuzzing in order to determine if the protocol correctly handles negative values calculated for the fee growth inside.

28. Functions missing the impure specifier

Severity: **Informational**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-TONCO-28
Target: `contracts/pool/*`	

Description

Multiple functions that update state or are able to throw errors are missing the `impure` specifier. While we did not discover any functions that would be removed by the compiler, adding the specifier will help prevent security issues that could be introduced during further development. The functions missing the specifier are listed below:

- `getTickAtSqrtRatio` (`contracts/pool/ticks.func#L77–L79`)
- `getSqrtRatioAtTick` (`contracts/pool/ticks.func#L27–L29`)
- `checkTickValidity` (`contracts/pool/amm.func#L11`)
- `getNextSqrtPriceFromOutput` (`contracts/pool/liquidity.func#L241–L243`)
- `getNextSqrtPriceFromInput` (`contracts/pool/liquidity.func#L218–L220`)
- `getNextSqrtPriceFromAmount1RoundingDown` (`contracts/pool/liquidity.func#L175–L177`)
- `getNextSqrtPriceFromAmount0RoundingUp` (`contracts/pool/liquidity.func#L119–L121`)
- `getAmount0DeltaR` (`contracts/pool/liquidity.func#L24–L26`)
- `swapOperationAllowed` (`contracts/pool/operations/pool_swap.func#L3`)
- `reforgeOperationAllowed` (`contracts/pool/operations/pool_reforge.func#L11`)

Recommendations

Short term, add the `impure` specifier to all functions that can throw errors or modify state.

Long term, ensure that access control helper functions are thoroughly tested in order to easily verify the compiler never removed the functions due to a missing specifier.

A. Vulnerability Categories

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

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

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

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

B. Code Maturity Categories

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

Code Maturity Categories	
Category	**Description**
Arithmetic	The proper use of mathematical operations and semantics
Auditing	The use of event auditing and logging to support monitoring
Authentication / Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Cryptography and Key Management	The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution
Decentralization	The presence of a decentralized governance structure for mitigating insider threats and managing risks posed by contract upgrades
Documentation	The presence of comprehensive and readable codebase documentation
Low-Level Manipulation	The justified use of inline assembly and low-level calls
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage
Transaction Ordering	The system's resistance to transaction-ordering attacks

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

C. Code Quality Findings

This appendix contains findings that do not have immediate or obvious security implications. However, addressing them may enhance the code's readability and may prevent the introduction of vulnerabilities in the future.

- **Incorrect arithmetic in comment.** The comment at `contracts/pool/pool_storage.func#L243` states `1023 - 1016 = 15`, but the correct result is 7.
- **Inconsistent code formatting throughout the codebase.** The codebase contains excessive line breaks and inconsistent spacing. Consider using the format document feature on Tolk files with the official TON VS Code extension and manually formatting FunC files to improve code readability.
- **Unused return values in getter functions.** The `cell account_code`, `cell position_nft_code`, and `cell nftitem_content` return values are unused (`contracts/pool/get.func#L351`).
- **Refactoring needed for `get_nft_content` function.** The function contains unused variables and variable shadowing, and loads dictionaries multiple times unnecessarily (`contracts/pool/get.func#L386–L454`).
- **Incorrect function return value documented in comment.** The comment at `contracts/pool/amm.func#L28` indicates three return values, but the function returns only two values of type `(int, int)`.
- **Use of hard-coded workchain values instead of defined constant.** The `calculatePoolAddress` function uses a hard-coded workchain value instead of the `WORKCHAIN` constant defined in `contracts/common/params.tolk#L1`. This hard-coded value appears in multiple locations: `contracts/pool/pool_utils.tolk#L109`, `contracts/account_tolk/account_utils.func#L32`, and `contracts/router_tolk/get.tolk#L45`.
- **Typographical error in variable name.** The code uses chached instead of cached (`contracts/pool/pool_storage.func#L376–L390`).
- **`pool::nft_items_active` counter never decremented.** The counter is incremented when initializing the pool but is never decremented, leading to an inaccurate count of active NFT items (`contracts/pool/operations/pool_create.func#L188`, `contracts/pool/operations/pool_reforge.func#L309`).

- **Incorrect comment about bit length.** The comment at `contracts/pool/pool_storage.func#L270–L274` incorrectly states that `addr_slice` holds 534 bits, but it actually holds 267 bits multiplied by 3.
- **Unreachable branch in `load_tick` function.** The function contains a branch checking for a bit length of 896, but this condition can never be true since the cached price is always saved to the cell (`contracts/pool/pool_storage.func#L365–L374`).
- **Misleading constant name.** The `MESSAGE_TAG` constant should be renamed to `BOUNCED_TAG` to accurately reflect its purpose of checking if a message resulted from a bounce (`contracts/pool/pool.func#L51–L54`).
- **Unused variables in account contract.** Two unused variables `MIN_TICK` and `MAX_TICK` can be removed from the account contract (`contracts/account_tolk/account.tolk#L19–L20`).
- **Magic number used instead of named constant.** The code uses the magic number 1 instead of the `MESSAGE_TAG` constant defined in `contracts/common/messages.func#L3` (`contracts/account_tolk/account.tolk#L33`).
- **Incorrect comment about pool state.** The comment at `contracts/pool/pool_utils.tolk#L33` incorrectly states that `pool_active` is part of the pool state when it is not.
- **Redundant calculation of pool order.** The code should use `calculatePoolStateInit` instead of `calculatePoolAddress` to avoid calculating the pool order twice (`contracts/router_tolk/operations/router_receive.tolk#L77–L78`).
- **Incorrect parameter names in getter function.** The `getFeeGrowthInside` getter uses incorrect parameter names, likely copied from the `getCollectedFees` function (`contracts/pool/get.func#L225–L229`).
- **Unused variable `fwdFee` in router contract.** The `fwdFee` variable is loaded but never used and can be skipped (`contracts/router_tolk/router.tolk#L48`).
- **Magic number used for operation code.** An operation at `contracts/pool/operations/pool_getter.func#L15` uses a magic number that does not match any defined constants. Using magic numbers is error-prone and should be avoided.
- **Incomplete comment.** The comment at `contracts/account_tolk/account_storage.tolk#L56` appears to be missing some words, making it unclear.

- **Duplicate validation check in mint operation.** The `rejectMint` check is performed twice even though the variable is not updated between checks (`contracts/pool/operations/pool_mint.func#L117–L134`).
- **Unused type definition.** The `PosAnsFeesStorage` type is defined but never used and can be removed (`contracts/common/position.tolk#L35`).
- **Pool sequence number is always zero in the pay-to operation.** The pool sequence number is always set to zero in the account `ROUTER_OPERATION_PAY_TO` message (`contracts/pool/operations/pool_burn.func#L95`).
- **Unused variables in burn operation.** The variables `s` and `r` returned from the `slice_bits_refs` function can be removed from the burn operation (`contracts/pool/operations/pool_burn.func#L130`).
- **Use of custom constant instead of standard library constant.** The code defines and uses a custom `MODE_CARRY_REMAINING_GAS` constant (`contracts/common/messages.func#L14`) instead of using the official `SEND_MODE_CARRY_ALL_REMAINING_MESSAGE_VALUE` constant from the standard library (`contracts_other/jetton-stable/stdlib_ext1.fc#L209–L210`).
- **Unnecessary variable redeclaration.** A variable is redeclared unnecessarily in the AMM logic (`contracts/pool/amm.func#L216-L228`).
- **Magic error code instead of named constant.** The `positionnft` contract uses a magic error code instead of the `WRONG_OP` constant defined in `contracts/common/errors.tolk#L5` (`contracts/positionnft_tolk/positionnft.tolk#L129`).
- **Unused function parameter.** The `int zeroForOne` parameter in the `crossTick` function is unused and can be removed (`contracts/pool/amm.func#L211`).
- **Inconsistent bit widths for fee growth accumulators.** The new tick format reduces `outerFeeGrowth0Token` and `outerFeeGrowth1Token` from 256-bit to 224-bit (`contracts/pool/pool_storage.func#L370–L381`), while `feeGrowthGlobal0X128` and `feeGrowthGlobal1X128` remain 256-bit (`contracts/pool/pool_storage.func#L331–L332`). The `crossTick` function computes `outer = global - outer` and writes the result back into 224-bit storage (`contracts/pool/amm.func#L227–L228`). Consider storing these values in a referenced cell to maintain full precision, with the head cell containing `cachedPrice(160) | liquidityTotal(256) | liquidityDelta(128)` and the reference cell containing `outerFeeGrowth0Token(256) | outerFeeGrowth1Token(256)`.
- **Incorrect error constant name in NFT transfer.** The error constant name is incorrect (copied from a different contract) but uses the correct error code of 708.

The constant should be named `INCORRECT_FORWARD_PAYLOAD` (`contracts/positionnft_tolk/operations/positionnft_transfer.tolk#L42`, `contracts/positionnft_tolk/errors.tolk#L13`).

- **Incorrect bit width in message serialization.** The code uses `.store_uint(7, 108)` when it should use `.store_uint(3, 107)`, calculated as 1 + 4 + 4 + 64 + 32 + 1 + 1 (`contracts/common/messages.func#L72`). Using `(7, 108)` has an extra bit that writes 1 into the `created_at` field. However, the validator will rewrite this field according to the TON message layout documentation, so this error has no practical impact.
- **Typographical error in gas constant.** The constant should be 21 instead of 211 (`contracts/common/gas.fc#L70–L71`).
- **Missing early return for null check.** The condition `if (payToBody.coins == null)` should return early when true to avoid processing invalid data (`contracts/router_tolk/operations/router_send.tolk#L26–L30`).
- **Misleading operation constant name.** The constant at `contracts/account_tolk/account.tolk#L63` has a misleading name and should be `ACCOUNT_OPERATION_PAY_TO` to accurately reflect its purpose.

D. Automated Testing Guidance

During the security review, we built a differential fuzzing test suite in order to compare the TONCO mathematics with the Uniswap v3 reference implementation. Our approach consisted of the following:

- Using Foundry as the fuzzing orchestrator
    - Each fuzz test is written as a Foundry stateless fuzz test. This allowed us to define the bounds and invariants in Solidity and run the Uniswap v3 functions natively. Each call to the TON contracts is made using Foundry's JSON and FFI utilities by sending POST requests to a locally running server and processing the response.
- Creating a testing harness smart contract in FunC that provides getter methods for each of the functions under test, along with the appropriate wrappers for easy handling in the TypeScript tests
- A simple Node.js server that uses Blueprint and Sandbox to compile the contract, run the appropriate function provided via the request, and return a response with the result along with any potential errors due to execution failures

The functions indicated in the Automated Testing section were compared under the following criteria:

- If the reference implementation reverts, the TONCO implementation is also expected to revert.
- The values returned by the TONCO implementation should be exactly equal to the values returned by the reference implementation.

This approach allowed us to differentially fuzz the two implementations even though the two blockchain implementations and the tooling used to test them are completely different. While the need to send asynchronous requests and wait for a response slowed down fuzzing speed, this approach is still valuable since the TON ecosystem does not have dedicated tooling for fuzzing TON smart contracts.

How to Run the Test Suite

To run the test suite, first run the Node.js server via the following command:

```
ts-node scripts/ton_math_server.ts
```

Figure D.1: Running the Node server

Additionally, the `--trace` flag can be added in order to inject debug statements into the Fift code of the harness and recompile the harness with the modified Fift code.

To run the actual tests, install Foundry and run the following command in the `foundry/` directory:

```
// Run all the tests
forge test --ffi -vv

// Run a specific test
forge test --mt name_of_test --ffi -vv
```

Figure D.2: Commands for running fuzz tests

How to Add Test Cases

When adding a new test case for a function that is currently not included in the Node.js server, make the following additions:

1. Create a unique endpoint as the entrypoint for the function call in the Node.js server; for instance, the `getNextSqrtPrice0` endpoint looks like this:

```
app.post('/getNextSqrtPrice0', async (req: Request<{}, {},
NextSqrtPriceRequest>, res: Response) => {
    try {
        const { sqrtP, L, amount, add } = req.body;
        const result = await callMathFunction('getNextSqrtPrice0', [
            safeBigInt(sqrtP), safeBigInt(L), safeBigInt(amount),
safeBigInt(add)
        ]);
        res.json(serializeResult(result));
    } catch (e) {
        res.status(400).json({ ok: false, error: e instanceof Error ?
e.message : String(e) });
    }
});
```

Figure D.3: The `getNextSqrtPrice0` endpoint

2. Create a wrapper function in the `TonMathCaller.s.sol` contract that will construct the JSON, make the request, and return the response:

```
function getNextSqrtPrice0(uint256 sqrtP, uint256 L, uint256 amount, bool add)
public returns (bool, uint256) {
    string memory jsonData = string(abi.encodePacked(
        '{"sqrtP":"', vm.toString(sqrtP),
        '","L":"', vm.toString(L),
        '","amount":"', vm.toString(amount),
        '","add":"', add ? "1" : "0",
        '"}'
    ));
```

```
    string memory response = makeRequest("getNextSqrtPrice0", jsonData);

    // Check if the response indicates success
    bool ok = response.readBool(".ok");
    if (!ok) {
        console.log("TON error: ", response.readString(".error"));
        return (false, 0);
    }
    return (true, response.readUint(".result"));
}
```

Figure D.4: The `getNextSqrtPrice0` Solidity wrapper function

3. Write the test case as a normal Foundry stateless fuzzing test:

```
function testFuzz_NextSqrtPrice0(uint160 sqrtP, uint128 L, uint256 amount,
bool add) public {
    sqrtP = uint160(bound(uint256(sqrtP), 1, type(uint160).max));
    L = uint128(bound(uint256(L), 1, type(uint128).max));
    amount = bound(amount, 1, type(uint256).max);

    bool uniSuccess;
    bool tonSuccess;
    uint160 uniResult;
    uint256 tonResult;

    // Compare with Uniswap implementation
    try this.wrapper_getNextSqrtPriceFromAmount0RoundingUp(sqrtP, L, amount,
add) returns (uint160 result) {
        uniSuccess = true;
        uniResult = result;
        console.log("uniResult", uint256(result));
    } catch {
        uniSuccess = false;
    }

    (tonSuccess, tonResult) = getNextSqrtPrice0(sqrtP, L, amount, add);
    assertEq(tonSuccess, uniSuccess, "TON <> Uniswap success mismatch");
    assertEq(uint256(tonResult), uint256(uniResult), "NextSqrtPrice0
mismatch");
}
```

Figure D.5: The `getNextSqrtPrice0` fuzz test

The Node.js server currently assumes that all tests are performed on a single harness contract; however, this can be easily extended by compiling and deploying any other contract and modifying the `callMathFunction` function to call functions on this contract. This method could be used to enable stateful fuzzing of TON contracts.

E. Out-of-Scope Issues

This appendix contains findings discovered in components that were out of scope for this review, as they were not fully developed. The findings are included in order to reduce the chance that similar issues will surface in the future development of these components:

- **Pool settings can be changed by anyone due to a race condition in the pool factory.** When a user initiates a pool deployment via the `pool_factory` contract, a `POOL_FACTORY_OPERATION_ORDER_INIT` message is sent to the `factory_order` contract, which saves the settings of the pool in the persistent state of this contract. This state is later passed back to the `pool_factory` contract once both master jetton wallets have responded with the router's jetton wallet addresses. Since the `POOL_FACTORY_OPERATION_ORDER_INIT` message can be executed multiple times, a malicious user can monitor the system for deployment of an honest pool and start the deployment of the same pool with different (malicious) settings. Since the first deployment has not been finalized yet, the `factory_order` contract will save the new pool settings and later pass these malicious settings to the pool factory, deploying the pool with incorrect settings. (`contracts/pool_factory/factory_order.func`)
- **The `factory_order` contract's TON balance can be drained by repeated execution of the `OPERATION_TAKE_WALLET_ADDRESS` operation**, which calls the `accept_message` function. (`contracts/pool_factory/factory_order.func#L124–L136`).

F. Specification Guidelines

This section provides generally accepted best practices for and guidance on how to write design specifications.

A good design specification serves three purposes:

1. **It helps development teams detect bugs and inconsistencies in a proposed system architecture before any code is written.** Codebases that were written without adequate specifications are often littered with snippets of code that have lost their relevance as the system's design has evolved. Without a specification, it is exceedingly challenging to detect such code snippets and remove them without extensive validation testing.

2. **It reduces the likelihood that bugs will be introduced in implementations of the system.** In systems without a specification, engineers must divide their attention between designing the system and implementing it in code. In projects requiring multiple engineers, engineers may make assumptions about how another engineer's component works, creating an opportunity for bugs to be introduced.

3. **It improves the maintainability of system implementations and reduces the likelihood that future code changes will introduce bugs.** Without an adequate specification, new developers need to spend time "on-ramping," where they explore the code and understand how it works. This process is highly error-prone and can lead to incorrect assumptions and the introduction of bugs.

Low-level designs may also be used by test engineers to create property-based fuzz tests and by auditors to reduce the amount of time needed to audit a specific protocol component.

Specification Construction

A good specification must describe system components in enough detail that an engineer unfamiliar with the project can use the specification to implement those components. The level of detail required to achieve this can vary from project to project, but generally, a low-level specification will include the following details, at minimum:

- Details about how each system component (e.g., a contract or plugin) interacts with and relies on other components

- The actors and agents that participate in the system, the way they interact with the system, and their permissions, roles, authorization mechanisms, and expected known-good flows

- The expected failure conditions the system may encounter and the way those failures are mitigated, including failures that are mitigated automatically
- Specification details for each function that the system will implement, which should include the following:
    - A description of the purpose of the function and its intended use
    - A description of the function's inputs and the various validations that are performed against each input
    - Any specific restrictions on the function's inputs that are not validated or obvious
    - Any interactions between the function and other system components
    - The function's various failure modes, such as failure modes for queries to a Chainlink oracle for a price (e.g., stale price, disabled oracle)
    - Any authentication/authorization required by the function
    - Any function-level assumptions that depend on other components behaving in a specific way

In addition, specifications should use standardized RFC-2119 language as much as possible. This language pushes specification authors toward a writing style that is both detailed and easy to understand. One relevant example is the ERC-4626 specification; this specification uses RFC-2119 language and provides enough constraints on implementers so that a vault client for a single implementation may be used interchangeably with other implementations.

Example: Interaction Specification

An interaction specification is used to describe how the components of the system depend on each other. It includes a description of the other components that the system interacts with, the nature of those interactions, expected behavior or dependencies, and access relationships.

Note that a diagram can often be a helpful aid for modeling component interactions, but it should not be used to substitute a textual description of the component's interactions. Part of the goal of a specification is to help derive a list of properties that can be explicitly tested, and deriving properties from a diagram is much more challenging and error-prone than from a textual specification.

Here is an example of an interaction specification:

MyERC721 interacts with the following contracts:

- *ManagerRegistry: MyERC721 queries this contract to determine whether a caller is an administrator. This is used in all of the functions that define the administratorOnly modifier. The appManager address is configurable via the setAppManagerAddress function.*
- *MyERC721Handler: MyERC721 calls this contract through the _beforeTokenTransfer hook whenever tokens are transferred, minted, or burned. The MyERC721Handler contract must implement the checkRules function, which is used to check application- and token-specific rules. The MyERC721Handler is configurable via the connectHandlerToToken function.*

The following contracts interact with MyERC721:

- *[Example contract]*

The following actors interact with MyERC721:

- *[Example actor]*

Example: Function Specification

A function specification is used to describe the purpose and intended behavior of the function. It includes a description of the purpose and intended use of the function, the input parameters and all the validation performed on the inputs, any assumptions or implicit restrictions, the interactions between the function and system components, the failure modes, and the authentication required by the function.

Here is an example of a specification for the `connectHandlerToToken(address _deployedHandlerAddress)` function:

> *The connectHandlerToToken(address _deployedHandlerAddress) function is called by an administrator to connect the token contract to a previously deployed token handler contract.*
>
> *If the caller is not an administrator, connectHandlerToToken() **must** revert.*
>
> *If the caller is an administrator and the provided address is not the zero address, connectHandlerToToken() **must not** revert.*
>
> *If the caller is an administrator and the provided address is not the zero address, connectHandlerToToken() **must** set the stored handler address and emit the HandlerConnected event.*

A complementary technique for defining a function specification, which can be especially useful for defining test cases, is the branching tree technique (proposed by Paul Berg), which is a tree-like structure based on all of the execution paths, the contract state or function arguments that lead to each path, and the end result of each path.

Figure F.1 shows an example of a branching tree specification for the `connectHandlerToToken()` function:

```
connectHandlerToToken(address _deployedHandlerAddress)
├── when the caller is not an admin
│   └── it should revert
└── when the caller is an admin
    ├── when the _deployedHandlerAddress is the same as the current handler address
    │   ├── it should re-set the handlerAddress state variable
    │   ├── it should re-set the handler state variable
    │   └── it should emit a {HandlerConnected} event
    └── when the _deployedHandlerAddress is not the same as the current handler
address
        ├── when _deployedHandlerAddress is the zero address
        │   └── it should revert
        └── when the _deployedHandlerAddress is not the zero address
            ├── it should set the handlerAddress state variable
            ├── it should set the handler state variable
            └── it should emit a {HandlerConnected} event
```

Figure F.1: An example branching tree specification for the `connectHandlerToToken` function

This type of specification can be useful when developing unit tests, as it makes it easy to identify the execution paths, conditions, and edge cases that need to be tested.

Example: Constraints Specification

A constraints specification is used to describe all of the constraints that are explicitly or implicitly applied to a contract, function, or component.

Here is an example of a constraints specification:

> *A `MyERC721` contract is deployed by application administrators.*
>
> *`MyERC721` has two constraints that limit the contract's operation: `Pausable` and the `MyERC721Handler` address. `Pausable` is used to stop the contract when there is an issue with the overall system, and the `MyERC721Handler` address is used to check that a transfer, mint, or burn operation does not violate the rules of the protocol. If the address is not defined or is misconfigured, the `MyERC721` contract will revert on any transfer, mint, or burn operation.*

G. Fix Review Results

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

From January 12 to January 16, 2026, Trail of Bits reviewed the fixes and mitigations implemented by the TONCO team for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue.

The TONCO team implemented fixes in the `algebra-ton-contracts` repository, with the primary changes consolidated in commit `fd2015ce29144ee0769249ad8559ca156d22ca93` (tag `v1.6-postaudit-fixes`). Due to the need for extended bounce support to properly handle jetton refunds when messages bounce from nonexistent pools (TOB-TONCO-16), the team upgraded to Tolk 1.2 during the fix process. This upgrade enabled proper bounce handling across multiple contract types but required some fixes to be implemented across multiple commits. All three high-severity findings—the fee calculation mismatch (TOB-TONCO-1), proxy TON theft issue (TOB-TONCO-6), and router TON drainage issue (TOB-TONCO-21)—were successfully resolved.

In summary, of the 28 issues described in this report, TONCO has resolved 21 issues, has partially resolved four issues, and has not resolved the remaining three issues. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Severity	Status
1	Fee calculation mismatch in PositionNFT burn operation causes incorrect fee growth tracking for token1	High	Resolved
2	owner_address can be spoofed to bypass the pool lock in the swapOperation function	Medium	Resolved
3	timelock_delay cannot be updated	Low	Resolved
4	ALM mint path sends ALM address instead of user address in message body	Undetermined	Resolved

5	Occupied ticks guard blocks addition of liquidity to existing ticks and allows MAX_USER_TICKS overflow	Medium	Resolved
6	Proxy TON balance of the router can be stolen	High	Resolved
7	Missing slippage protection for mint and burn orders	Medium	Unresolved
8	price_sqrt can be manipulated by swapping through an empty pool	Medium	Resolved
9	Zero-liquidity positions can be minted and deposited into an account	Informational	Resolved
10	Missing zero address check in the reforgeOperation function can lead to loss of funds	Low	Partially Resolved
11	Multihop shortcut allows a position to be minted through the router contract's account	Informational	Partially Resolved
12	Malformed multihop cell can lead to jetton loss	Medium	Resolved
13	Router will keep the jettons if the transfer notification contains an unsupported operation	Low	Resolved
14	Pool reinitialization risks	Low	Partially Resolved
15	Return value of modifyPosition is ignored inside burnPosition	Informational	Resolved
16	Router does not check for pool existence	Informational	Resolved
17	Lack of transparency in timelocked code updates	Informational	Resolved
18	Manual balance calculation instead of raw_reserve	Informational	Partially Resolved

19	Users are able to route positions to the ALM	Undetermined	Resolved
20	Excess TON refunded to the wrong address in swaps	Informational	Resolved
21	The router's TON balance can be drained via negative amount calculation	High	Resolved
22	Overflow in getMaxLiquidityForAmount0Precise can result in fund loss	Medium	Resolved
23	Incomplete handling of swap exceptions	Informational	Resolved
24	Depositing more than four positions causes failure	Informational	Resolved
25	Incorrect price caching corrupts legacy tick format	Informational	Resolved
26	Rounding difference between Uniswap v3 and TONCO amount delta calculations	Undetermined	Unresolved
27	Potential overflow in fee growth computation	Undetermined	Unresolved
28	Functions missing the impure specifier	Informational	Resolved

Detailed Fix Review Results

TOB-TONCO-1: Fee calculation mismatch in PositionNFT burn operation causes incorrect fee growth tracking for token1
Resolved in commit d86cd6c9. The incorrect variable assignment, `position0.growthInside1LastX128 = newFees.growthInside0LastX128`, has been replaced with the correct assignment, `position0.growthInside1LastX128 = newFees.growthInside1LastX128`, restoring proper fee calculation behavior for token1.

TOB-TONCO-2: owner_address can be spoofed to bypass the pool lock in the swapOperation function
Resolved in commit 8eb46b23. A new field that stores the authenticated source of the transaction has been introduced to the `POOL_SWAP` message sent by the router. The access control checks for lock override have been updated to reference this trusted field instead of the user-supplied `owner_address`.

TOB-TONCO-3: timelock_delay cannot be updated
Resolved for new deployments via `state-init` in commit b6c6ec4b. The client does not plan to make `timelock_delay` mutable post-deployment, but the default delay is now enforced in code at deployment time: the pool's `state-init` encodes a default timelock delay of 86,400 seconds (1 day).

The client provided the following context for this finding's fix status:

> *We have no plans to make timelock mutable. We have added a timelock value check to the deploy checklist, so we have an expected value of 1-2 days before proceeding with the deploy.*

TOB-TONCO-4: ALM mint path sends ALM address instead of user address in message body
Resolved in commit 3d5edee1. The message structure between the router and pool has been reworked to include space for additional addresses. This allows liquidity to be added to ALM, `Arbiter`, and `UserAccount` either directly by the user or on behalf of another user in a consistent manner.

TOB-TONCO-5: Occupied ticks guard blocks addition of liquidity to existing ticks and allows MAX_USER_TICKS overflow
Resolved in commits 8e030a20 and a5df9ec5. The pool now computes the post-operation occupied tick count before applying state changes, preventing the counter from incrementing past the tick limit. The client is planning to add related tests later in a separate commit.

TOB-TONCO-6: Proxy TON balance of the router can be stolen
Resolved in commit 8db1d079. Additional validation checks have been added to the multihop shortcut logic to verify that when the receiver is the `proxyTonWallet`, the jetton

wallet used in the swap is also the `proxyTonWallet`, preventing attackers from draining proxy TON through malicious jetton swaps.

TOB-TONCO-7: Missing slippage protection for mint and burn orders
Unresolved. The TONCO team considers this behavior to be by design. For mints, users can send more coins than required to mint a specified amount of liquidity, allowing them to control slippage. For burns, the client has acknowledged the issue and added it to its backlog for future consideration when the revised burn workflow is introduced.

The client provided the following context for this finding's fix status:

> *There is a set of tools to control for slippage in case of minting, the user can send more coins than is required to mint a specified amount of liquidity, thus agreeing to the predefined slippage. Burning in the current design is irreversible - if it started - it should succeed. We have added this feature to the backlog.*

TOB-TONCO-8: price_sqrt can be manipulated by swapping through an empty pool
Resolved in commit `f9806c06`. A check has been added for the specific case of a zero-value `pool::occupied_ticks`. This condition causes the swap to revert with a distinct error, preventing empty-pool price manipulation. The Pool Creator and Controller are allowed to override this check to simplify deployment and initial setup. Note that the override is granted to the `POOL_ROLE_CONTROLLER` and `POOL_ROLE_CREATOR` roles (not `POOL_ROLE_ADMIN`).

TOB-TONCO-9: Zero-liquidity positions can be minted and deposited into an account
Resolved in commit `cec31b81`. Zero-liquidity checks have been added in the NFT deposit message processing and in the pool reforge method during pass-through, ensuring zero-liquidity positions cannot be created through these paths.

TOB-TONCO-10: Missing zero address check in the reforgeOperation function can lead to loss of funds
Partially resolved in commit `35b5d723`. The initialization of default addresses for the ALM, `Arbiter`, `Oracle`, and `Creator` has been updated to properly handle null addresses, and the address for a nonexistent ALM now defaults to null in wrappers and during creation via the router. However, a remaining edge case can orphan positions when `action_target` is set to `ACTION_TARGET_ALM` (1) but ALM routing is not actually enabled (the ALM is unset or `FLAG_USER2ALM_MASK` is disabled). In this scenario, `reforgeOperation` sets `pay_to_target` to `role_alm` only when the ALM is non-null and the caller is authorized, but it still packs positions into `alm_positions` whenever `action_target` is set to `ACTION_TARGET_ALM`, causing those positions to be skipped by the NFT minting loop. Since `pay_to_target` remains set to the router in this case, the positions are sent to the router's `payToOperation`, which ignores them (it only processes coins), resulting in orphaned positions. The TONCO team has been informed of this remaining edge case.

TOB-TONCO-11: Multihop shortcut allows a position to be minted through the router contract's account
Partially resolved in commit `fce08874`. The multihop shortcut processing has been updated to explicitly block `POOL_OPERATION_FUND_ACCOUNT`. However, the client acknowledges that when the shortcut is not active (when using a regular multihop without the shortcut), this problem remains, as it is still possible to create positions for any contract that allows transfer of coins with user-controlled payloads.

TOB-TONCO-12: Malformed multihop cell can lead to jetton loss
Resolved in commits `3d4a12f8`, `e1acf562`, `a4147b63`, and `9f18a9f4`. A lightweight structural parsing check has been added. If the payload cannot be parsed correctly, a parsing error triggers an exception and the refund occurs automatically.

TOB-TONCO-13: Router will keep the jettons if the transfer notification contains an unsupported operation
Resolved in commit `19176a60`. The entire message-processing flow is now wrapped in a try-catch block. If the funds cannot be processed correctly, they will be refunded to the user.

TOB-TONCO-14: Pool reinitialization risks
Partially resolved in commit `67c0d0a3`. Additional checks have been added for `tick_spacing` reinitialization to only allow a smaller tick spacing that evenly divides the previous value when positions already exist. However, the recommendations regarding fee percentage upper bounds and moving admin role updates to a two-step process were not implemented.

The client provided the following context for this finding's fix status:

> *We think that allowing fees up to 100% just gives additional flexibility in controlling the pool. The Admin Role gives no unique and additional rights to its owner, so temporary loss of access to it doesn't pose business/mission-critical risks.*

TOB-TONCO-15: Return value of modifyPosition is ignored inside burnPosition
Resolved in commit `bcb57f90`. An additional check has been added to validate the return value of `modifyPosition` in the burn path, ensuring consistent error handling across mint and burn operations.

TOB-TONCO-16: Router does not check for pool existence
Resolved in commits `b82cc4da`, `7b7efc64`, `3d69e477`, and `8f871c4f`. The bounce flag has been added to messages sent to the pool, and extended bounce processing has been implemented for swaps, mints/funds, and position deposits using Tolk 1.2's extended bounce support. This ensures jettons are refunded when messages bounce from nonexistent pools.

TOB-TONCO-17: Lack of transparency in timelocked code updates
Resolved in commit 34c62b54. Getter methods have been added to allow users and integrators to check whether a timelock action is currently pending and retrieve pending code update information.

Note that the router getter comment is out of sync with the actual return order and should be updated.

TOB-TONCO-18: Manual balance calculation instead of raw_reserve
Partially resolved in commits 1ac6bb30 (`Router`) and cbee57b2 (`Account`). The `Router` and `Account` contracts have been updated to use `raw_reserve()`. However, the `PositionNFT` code was intentionally not changed to keep it as close to the reference NFT code as possible.

TOB-TONCO-19: Users are able to route positions to the ALM
Resolved in commit b304f073. A flag (`FLAG_USER2ALM_MASK`) has been introduced that disables the ability for unprivileged users to route positions to the ALM in production environments. This functionality is retained for testing purposes but can be disabled via the flag.

TOB-TONCO-20: Excess TON refunded to the wrong address in swaps
Resolved in commit 8eb46b23. The swap message processing in the pool now sets `excess_receiver` to `fromUser` if it was not explicitly set in the `action_cell`, ensuring excess TON is refunded to the swap originator regardless of swap direction.

TOB-TONCO-21: The router's TON balance can be drained via negative amount calculation
Resolved in commit 089ffe55. Validation has been added to ensure that `tonAmount` is nonnegative before proceeding with message sending, preventing the router from spending its own TON on behalf of users who have not provided sufficient funds.

TOB-TONCO-22: Overflow in getMaxLiquidityForAmount0Precise can result in fund loss
Resolved in commits 97414bd3 and 487c0358. Following our recommendation, the implementation now uses `muldiv` to take advantage of 513-bit precision and prevent overflow in the multiplication step.

Note that the production mint "max liquidity" path (`computeLiquidity` → `getMaxLiquidityForAmounts`) no longer calls `getMaxLiquidityForAmount0Precise`; it uses the imprecise `getMaxLiquidityForAmount0/1` variants that now rely on `muldiv`. But `getMaxLiquidityForAmount0Precise` itself still contains the overflow-prone triple multiplication and should not be used in mint logic unless rewritten safely.

TOB-TONCO-23: Incomplete handling of swap exceptions
Resolved in commit `6a257ff6`. Additional error processing has been added to handle all unexpected exceptions in math operations, which are now reported as `RESULT_SWAP_INTERNAL_ERROR` (231).

TOB-TONCO-24: Depositing more than four positions causes failure
Resolved in commit `2b6488f0`. Extra positions beyond the four-position limit are now pushed via the `POOL_REFORGE` method to NFTs, preventing transaction failures when depositing more than four positions. The `deposit` method will be initially blocked after deployment until the corresponding UI is implemented.

TOB-TONCO-25: Incorrect price caching corrupts legacy tick format
Resolved in commit `aae81be2`. The legacy tick format has been completely removed from the codebase, eliminating the potential for format corruption.

TOB-TONCO-26: Rounding difference between Uniswap v3 and TONCO amount delta calculations
Unresolved. The TONCO team has indicated that this issue is still under investigation.

TOB-TONCO-27: Potential overflow in fee growth computation
Unresolved. The TONCO team has indicated that this issue is still under investigation, though it believes the condition is practically unreachable.

TOB-TONCO-28: Functions missing the impure specifier
Resolved in commit `b2bfbbdb`. The `impure` specifier has been added to the identified functions that can throw errors or modify state.

H. Fix Review Status Categories

The following table describes the statuses used to indicate whether an issue has been sufficiently addressed.

Fix Status	
Status	**Description**
Undetermined	The status of the issue was not determined during this engagement.
Unresolved	The issue persists and has not been resolved.
Partially Resolved	The issue persists but has been partially resolved.
Resolved	The issue has been sufficiently resolved.

About Trail of Bits

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

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

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

We specialize in software testing and code review assessments, supporting client organizations in the technology, defense, blockchain, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, Uniswap, Solana, Ethereum Foundation, Linux Foundation, and Zoom.

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

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

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

Trail of Bits performed all activities associated with this project in accordance with a statement of work and an agreed-upon project plan.

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

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