Security Review Report
NM-0544A Piltover

(June 11, 2025)

Contents

1 Executive Summary

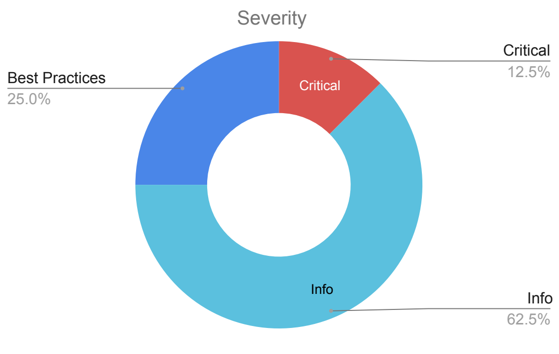
This document presents the results of a security review conducted by Nethermind Security for Piltover. **Piltover** is a system developed through a collaboration between Karnot and Cartridge. It is designed to extend the Starknet ecosystem by facilitating the creation and operation of Layer 3 (L3) solutions, known as Appchains, on top of the Starknet Layer 2 (L2). It serves a role analogous to Starknet's L1 core contract but is tailored for the L2/L3 relationship, enabling the recording and verification of L3 state transitions directly on Starknet.

The core of the Piltover system involves the `Appchain` contract, its central on-chain component on Starknet, which manages the L3's canonical state, cross-layer messaging, and essential configurations. An off-chain `Orchestrator` service is responsible for processing L3 data, generating cryptographic proofs of state transitions through StarknetOS (SNOS) and a Layout Bridge program, and submitting these proofs for verification. The integrity of L3 state is ensured by leveraging the Herodotus `FactRegistry` and Integrity Verifier for on-chain proof validation before state updates are finalized on the `Appchain` contract.

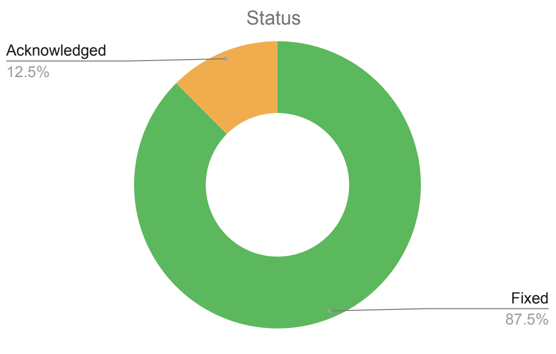
The audit comprises 979 lines of Cairo code. The audit was performed using (a) manual analysis of the codebase, (b) automated analysis tools, and (c) creation of test cases.

Along this document, we report 8 points of attention, where one is classified as `Critical`, and seven are classified as `Info` or `Best Practices` severity. The issues are summarized in Fig. 1.

This document is organized as follows. Section 2 presents the files in the scope. Section 3 summarizes the issues. Section 4 presents the system overview. Section 5 discusses the risk rating methodology. Section 6 details the issues. Section 7 discusses the documentation provided by the client for this audit. Section 8 presents the test suite evaluation and automated tools used. Section 9 concludes the document.



(a)



(b)

Fig. 1: Distribution of issues: Critical (1), **High** (0), **Medium** (0), **Low** (0), **Undetermined** (0), **Informational** (5), **Best Practices** (2).
Distribution of status: Fixed (7), **Acknowledged** (1), **Mitigated** (0), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	June 6, 2025
Final Report	June 11, 2025
Repository	keep-starknet-strange/piltover
Initial Commit	a7dc4141fd21300f6d7c23b87d496004a739f430
Final Commit	9970d44e729073de955d4e9ffa0441d20aab03a7
Documentation	book
Documentation Assessment	Low
Test Suite Assessment	Medium

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	src/snos_output.cairo	143	21	14.7%	14	178
2	src/interface.cairo	9	20	222.2%	1	30
3	src/appchain.cairo	164	47	28.7%	31	242
4	src/fact_registry.cairo	57	7	12.3%	6	70
5	src/lib.cairo	46	5	10.9%	10	61
6	src/config/component.cairo	91	26	28.6%	14	131
7	src/config/interface.cairo	16	61	381.2%	8	85
8	src/state/component.cairo	58	18	31.0%	11	87
9	src/state/interface.cairo	7	16	228.6%	2	25
10	src/components/onchain_data_fact_tree_encoder.cairo	29	31	106.9%	10	70
11	src/messaging/types.cairo	12	3	25.0%	2	17
12	src/messaging/component.cairo	279	96	34.4%	54	429
13	src/messaging/interface.cairo	31	100	322.6%	7	138
14	src/messaging/hash.cairo	37	32	86.5%	6	75
	Total	**979**	**483**	**49.3%**	**176**	**1638**

3 Summary of Issues

	Finding	Severity	Update
1	The `update_state(...)` can be blocked by a malicious actor	Critical	Fixed
2	Excessive `snos_output` size or message count can lead to `update_state(...)` failure and DoS	Info	Acknowledged
3	Missing validation of constituent program hash in `layout_bridge_output`	Info	Fixed
4	State can be updated with KZG data availability without verification	Info	Fixed
5	Subsequent calls to `start_message_cancellation(...)` will overwrite the previous cancellation time	Info	Fixed
6	The `DataAvailabilityFact` elements are not validated in the `update_state(...)` function	Info	Fixed
7	Missing input validation of the StarknetOS output fields	Best Practices	Fixed
8	Unused code	Best Practices	Fixed

4 System Overview

The Piltover system is designed to extend the Starknet ecosystem by enabling the creation and operation of Layer 3 (L3) solutions, often referred to as Appchains. It functions analogously to the Starknet core contract on Layer 1 (L1) but is specifically tailored for the Layer 2 (L2) / Layer 3 relationship within the Starknet environment. Piltover's primary role is to record and verify the state transitions of the L3 Appchains on the L2 Starknet, leveraging the Herodotus Integrity Verifier for cryptographic proof verification. An off-chain entity, the Orchestrator, is responsible for generating the proofs of L3 state transitions and submitting them to the appropriate Starknet contracts to finalize L3 state updates on L2.

4.1 Appchain Contract

The `Appchain` contract, deployed on Starknet, is the central on-chain component of the system. It serves as the L3 Appchain's anchor and messaging interface on L2. It is sometimes referred to as the Piltover contract.

- **State Management**: It maintains the canonical state of the L3 Appchain, including the latest block number, block hash, and state root. This state is updated via the `update_state(...)` function when a valid proof of state transition is provided.
- **Messaging Hub**: It facilitates bidirectional messaging between Starknet (L2) and the Appchain (L3). Users interact with `send_-message_to_appchain(...)` to dispatch messages to L3, and consume messages originating from L3 via `consume_message_from_-appchain(...)`.
- **Configuration Storage**: It stores critical operational parameters that are configurable by a designated Operator. These include the bootloader program hash, StarknetOS (SNOS) configuration and program hash, and the layout bridge program hash, which are essential for the state verification process.
- **Data Availability**: In the current implementation, the data required to reconstruct the Appchain's state (state transition data) is submitted as part of the StarknetOS program output within the calldata of the `update_state(...)` transaction on Starknet.

4.2 The Orchestrator

The `Orchestrator` is a critical off-chain service responsible for managing the L3 state finalization process. This entity typically assumes the "Operator" role, granting it specific privileged permissions on the `Appchain` contract.

- **Proof Generation Pipeline**: The Orchestrator gathers block data from the L3 Appchain's Sequencer, executes the StarknetOS (SNOS) to process L3 transactions and compute state transitions. The resulting SNOS output is then processed through a Layout Bridge program to generate a proof that is verifiable on Starknet.
- **Fact Submission**: It submits the output of the Layout Bridge program (a recursive proof) to the Herodotus Fact Registry contract on Starknet L2. This step is for on-chain verification of the L3 state transition.
- **State Update Initiation**: Following successful proof verification by the Fact Registry, the Orchestrator calls the `update_state(...)` function on the `Piltover Contract`. This action finalizes the L3 block(s) by updating the Appchain's state representation on L2.
- **System Configuration**: As the Operator, the Orchestrator is authorized to configure key operational parameters and program hashes stored within the `Piltover Contract`.

4.3 State Verification and Update

The integrity of L3 Appchain state transitions is ensured through a multi-stage cryptographic proof and verification process, which ends with a state update on the `Appchain` contract on Starknet.

- **StarknetOS (SNOS program) Execution**: The Orchestrator runs the SNOS program using L3 transaction data. The output generated by SNOS (the `SNOS output`) encapsulates the results of the L3 state transition.
- **Layout Bridge Program**: As the raw `SNOS output` may not be directly verifiable by the existing verifiers on Starknet, it is first processed by a `Layout Bridge Program`. This program generates a recursive proof, termed the `Layout Bridge Output`, which is structured specifically for on-chain verification by the Herodotus Integrity Verifier.
- **Herodotus Fact Registry**: The `Layout Bridge Output` proof is submitted to the Herodotus `FactRegistry` contract on Starknet. This contract uses the Herodotus Integrity Verifier to validate the proof. Upon successful verification, data pertaining to the proven fact is stored within the `FactRegistry`. The stored data includes a `fact_hash` (representing `poseidon_hash(program_hash, output_hash)`) and details of its verification such as proof `security_bits`.
- **`update_state(...)` Invocation**: Once a valid fact is registered in the `FactRegistry` (and meets criteria such as minimum `security_-bits`), the Orchestrator calls `update_state(...)` on the `Appchain` contract. This function confirms the proof's verification against the `FactRegistry` and, if valid, updates the L3's official state (block number, block hash, state root) on L2 and processes any associated cross-layer messages.

4.4 Cross-Layer Messaging

The Piltover system facilitates communication between the Starknet L2 and the L3 Appchain, operating in a manner similar to the messaging mechanism between Starknet and its Layer 1 (Ethereum).

- **Starknet (L2) to Appchain (L3)**:
  - Users initiate messages by calling the `send_message_to_appchain(...)` function on the `Piltover Contract` on L2.
  - These messages are initially marked with a `PENDING` status.
  - When the Orchestrator calls `update_state(...)`, if the L3 Sequencer has included these messages in the StarknetOS program output (signifying their successful processing and inclusion on L3), they are transitioned to a `SEALED` status within the `Piltover Contract`. If a transaction corresponding to a message reverted on L3, it will not be included in the output and will remain `PENDING`.
  - Messages in the `PENDING` state can be cancelled by the sender. This is a two-step process: first, an initiation of the cancellation, followed by the actual cancellation call after a predetermined delay period (e.g., 5 days), provided the message has not been `SEALED` in the interim.
- **Appchain (L3) to Starknet (L2)**:
  - Users on the L3 Appchain send messages to L2 by utilizing the `send_message_to_L1` syscall (note: "L1" in this context refers to the L2 Starknet from the perspective of the L3).
  - The L3 Sequencer collects these messages. They are then relayed to Starknet as part of the data submitted by the Orchestrator during an `update_state(...)` operation.
  - Upon the successful execution of `update_state(...)` on L2, these L3-to-L2 messages are marked as ready for consumption on Starknet.
  - The designated recipient address on L2 must then manually call the `consume_message_from_appchain(...)` function on the `Piltover Contract` to process and claim the message. Consumption is restricted to the specified recipient.

5 Risk Rating Methodology

The risk rating methodology used by Nethermind Security follows the principles established by the OWASP Foundation. The severity of each finding is determined by two factors: **Likelihood** and **Impact**.

Likelihood measures how likely the finding is to be uncovered and exploited by an attacker. This factor will be one of the following values:

a) **High**: The issue is trivial to exploit and has no specific conditions that need to be met;

b) **Medium**: The issue is moderately complex and may have some conditions that need to be met;

c) **Low**: The issue is very complex and requires very specific conditions to be met.

When defining the likelihood of a finding, other factors are also considered. These can include but are not limited to motive, opportunity, exploit accessibility, ease of discovery, and ease of exploit.

Impact is a measure of the damage that may be caused if an attacker exploits the finding. This factor will be one of the following values:

a) **High**: The issue can cause significant damage, such as loss of funds or the protocol entering an unrecoverable state;

b) **Medium**: The issue can cause moderate damage, such as impacts that only affect a small group of users or only a particular part of the protocol;

c) **Low**: The issue can cause little to no damage, such as bugs that are easily recoverable or cause unexpected interactions that cause minor inconveniences.

When defining the impact of a finding, other factors are also considered. These can include but are not limited to Data/state integrity, loss of availability, financial loss, and reputation damage. After defining the likelihood and impact of an issue, the severity can be determined according to the table below.

		Severity Risk		
Impact	**High**	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Info/Best Practices	Low	Medium
	Undetermined	Undetermined	Undetermined	Undetermined
		Low	**Medium**	**High**
		Likelihood		

To address issues that do not fit a High/Medium/Low severity, Nethermind Security also uses three more finding severities: **Informational**, **Best Practices**, and **Undetermined**.

a) **Informational** findings do not pose any risk to the application, but they carry some information that the audit team intends to pass to the client formally;

b) **Best Practice** findings are used when some piece of code does not conform with smart contract development best practices;

c) **Undetermined** findings are used when we cannot predict the impact or likelihood of the issue.

6 Issues

6.1 [Critical] The `update_state(...)` can be blocked by a malicious actor

File(s): `src/appchain.cairo`

Description: The `update_state(...)` function ensures that the `layout_bridge_output` passed as an argument is verified through the fact registry. The operator must first pass the Stark proof to the fact registry to prove the fact and then call `update_state(...)`. It also ensures that the proof is verified with at least 50 security bits.

The issue lies in the verification process within the `update_state(...)` function. When all verifications for a given fact are retrieved, the code only checks the security bits of the verification at the 0th index of the returned array.

```
fn update_state(
    ref self: ContractState,
    snos_output: Span<felt252>,
    layout_bridge_output: Span<felt252>,
    onchain_data_hash: felt252,
    onchain_data_size: u256,
) {
    // ...
    let program_info = self.config.program_info.read();
    // ...
    let output_hash = poseidon_hash_span(layout_bridge_output);
    // ...
    let fact = poseidon_hash_span(
        array![program_info.bootloader_program_hash, output_hash].span(),
    );
    let verifications = IFactRegistryDispatcher {
        contract_address: self.config.get_facts_registry(),
    }
        .get_all_verifications_for_fact_hash(fact);

    if verifications.len() == 0 {
        core::panic_with_felt252(errors::NO_FACT_REGISTERED)
    };
    // @audit-issue It always checks the element at 0th index
    assert!(*verifications.at(0).security_bits > 50);
    // ...
}
```

This means that if a malicious actor registers a valid fact with fewer than `50` security bits before the legitimate operator registers the same fact with the required `50` or more security bits, the `update_state(...)` function will revert. This is because the `assert!(*verifications.at(0).security_bits > 50)` check will fail, as it will evaluate the low-security verification submitted by the attacker.

The impact of this vulnerability is that a malicious actor can effectively grief the operator by front-running the state update and registering a low-security verification for a valid fact, preventing the legitimate state transition. This could lead to a denial of service for state updates, potentially halting critical operations.

Given the asynchronous nature of the sequencer and the orchestrator (operator), where the sequencer continuously produces blocks and the orchestrator submits state updates periodically, this vulnerability presents a significant risk. Since the sequencer's RPC endpoint for block data is public, an attacker has a window of opportunity to fetch this data, compute a proof, and submit it with lower security bits before the legitimate operator. This action not only blocks the orchestrator's `update_state(...)` call but can also necessitate a chain reorganization. If the sequencer has produced blocks ahead of the blocked state update, those blocks cannot be finalized, leading to instability and potential rollbacks in the appchain.

Recommendation(s): Consider revisiting the fact verification mechanism. Instead of relying on a specific order or index of registered facts, ensure that the system verifies if any submitted proof for the fact meets the required security threshold. This could involve iterating through all registered proofs for a given fact or utilizing a dedicated function within the fact registry that directly checks for a minimum security level.

Status: Fixed

Update from the client: This issue has been addressed in 82665b7e by using the `utils` library of `integrity` to check if any submitted proofs for the fact meets the security threshold.

6.2 [Info] Excessive `snos_output` size or message count can lead to `update_state(...)` failure and DoS

File(s): `src/appchain.cairo`, `src/messaging/component.cairo`

Description: The Piltover system's state is updated on Starknet via the `update_state(...)` function, which accepts the Starknet OS output (`snos_output`) as calldata. This `snos_output` can include messages to be relayed from the Appchain to Starknet, as well as messages to be processed for the Appchain itself. The processing of these messages occurs within the `update_state(...)` transaction's execution flow, primarily handled by functions within the `src/messaging/component.cairo` file.

Two primary issues can arise from the `snos_output` and its constituent message content, potentially leading to a Denial of Service (DoS) by causing the `update_state(...)` transaction to revert on Starknet:

1. **`snos_output` Calldata Size Exceeding Starknet Limits**: The entire `snos_output` is passed as calldata to the `update_state(...)` function. Starknet imposes a transaction calldata limit (currently `5000` felts, as per Starknet documentation). Specific sequencer implementations like Madara might also enforce stricter limits on parts of this calldata (e.g., Madara's default configuration for message segments). If the total size of `snos_output` exceeds these calldata limits, the `update_state(...)` transaction will fail;
2. **Message Processing Leading to Excessive Event Emissions**: The `snos_output` can trigger the processing of messages in both directions (to Starknet and to the Appchain). Both respective processing functions emit events, contributing to a cumulative event count for the `update_state(...)` transaction;

The `process_messages_to_starknet(...)` function handles messages directed from the Appchain to Starknet:

```
    // @audit Processes messages from snos_output to Starknet, emitting events.
    fn process_messages_to_starknet(
        ref self: ComponentState<TContractState>, messages: Span<MessageToStarknet>,
    ) {
        let mut messages = messages;

        loop {
            match messages.pop_front() {
                Option::Some(m) => {
                    let from = *m.from_address;
                    let to = *m.to_address;
                    let payload = *m.payload;

                    let message_hash = hash::compute_message_hash_appc_to_sn(from, to, payload);

                    // @audit-issue Event emission for messages to Starknet.
                    // @audit This contributes to the per-transaction event limit.
                    self.emit(MessageToStarknetReceived { message_hash, from, to, payload });
                    // ...
                Option::None => { break; },
            };
        };
    }
```

This function emits a `MessageToStarknetReceived` event for each message processed.

Similarly, messages intended for the Appchain (originating from Starknet and included in `snos_output` for state updates) are processed by `process_messages_to_appchain(...)`, which also emits events:

```
// @audit Processes messages from snos_output to the Appchain, also emitting events.
fn process_messages_to_appchain(
    ref self: ComponentState<TContractState>, messages: Span<MessageToAppchain>,
) {
    let mut messages = messages;

    loop {
        match messages.pop_front() {
            Option::Some(m) => {
                let from = *m.from_address;
                let to = *m.to_address;
                let payload = *m.payload;
                let selector = *m.selector;
                let nonce = *m.nonce;

                let message_hash = hash::compute_message_hash_sn_to_appc(
                    from, to, selector, payload, nonce,
                );
                // ...
                // @audit-issue Event emission for messages to the Appchain.
                // @audit This also contributes to the total events per transaction.
                self.emit(MessageToAppchainSealed {
                    message_hash, from, to, selector, payload, nonce
                });
            },
            Option::None => { break; },
        };
    };
}
```

This function emits a `MessageToAppchainSealed` event for each such message processed.

Starknet imposes a global limit of `1000` events that a single transaction can emit (as per Starknet documentation). If the combined number of messages (to Starknet and to Appchain) processed within a single `update_state(...)` call leads to the total event count from both functions exceeding this Starknet-defined limit, the entire `update_state(...)` transaction will revert.

While the loops within these functions do not have explicit iteration caps, the effective limitations are imposed externally by Starknet's transaction-level constraints. If an Appchain generates an `snos_output` that is either too large (calldata) or triggers the processing of too many messages cumulatively (event count), the `update_state(...)` call will fail, preventing state finalization.

The Piltover framework currently assumes upstream components (e.g., Appchain's sequencer/blockifier) will constrain `snos_output`. However, without intrinsic safeguards, misconfigurations or malicious `snos_output` could trigger these DoS conditions.

It might lead to a situation, where the Appchain becomes unable to finalize its blocks and prove its state on Starknet. This occurs because the `update_state(...)` transaction consistently reverts due to exceeding Starknet's calldata size limits or cumulative event emission limits from processing messages in both directions.

Recommendation(s): Consider revisiting the message processing mechanisms and implementing limits on the maximum number of messages that can be processed during a single state update. A possible mitigation might include changes at different levels of the tech stack, such as at the sequencer or blockifier level, or directly in the contract by splitting the message processing into batches.

Status: Acknowledged

Update from the client: As mentioned in the issue comment, the appropriate place to handle these limitations is at the sequencer level. Since Blockifier serves as the common execution layer for Starknet transactions, block limits must be configured to ensure that state updates are correctly processed by Piltover (which may differ from Starknet current limits).

Therefore, no changes will be made at the Cairo level to address this in the current state of Piltover. It is the responsibility of appchain sequencers to enforce constraints that prevent blocks from exceeding the limits that Piltover can handle for state updates.

6.3 [Info] Missing validation of constituent program hash in `layout_bridge_output`

File(s): `src/appchain.cairo`

Description: The `update_state(...)` function processes state updates by verifying proofs of Starknet OS (SNOS) executions. This involves a `bootloader_program` running a `layout_bridge_program`. The output of this, `layout_bridge_output`, is structured for verification by an external fact registry (Herodotus).

The `layout_bridge_output` span is understood to have a specific structure where various elements represent Poseidon hashes of programs or data. The element at index `2` is confirmed to be the `layout_bridge_program_hash` and the element at index `4` is the `snos_output_hash`; both are correctly validated. However, the element at index `3` of `layout_bridge_output`, which is understood to represent a program hash integral to the generation of this `layout_bridge_output` (potentially the bootloader program that ran the layout bridge, or another relevant program), is not validated against any expected value from the contract's configuration.

Given that the element at index `3` of `layout_bridge_output` represents a program hash crucial to the output's generation, the lack of its validation against an expected value is a concern. While the `fact` hash submitted to the Herodotus registry is calculated using the `program_info.bootloader_program_hash` from contract storage, a critical validation is missing: ensuring that the specific program hash present at `layout_bridge_output.at(3)` (which played a role in creating `layout_bridge_output`) matches an expected, legitimate program hash stored in the configuration. Without this check, if `layout_bridge_output` was generated using an unexpected or malicious intermediary program (whose hash would be at index `3`), the system might be misled. This could compromise the integrity of the state transition mechanism by allowing outputs from unintended program flows to be processed, even if other parts of the proof verification appear consistent.

```
fn update_state(
    ref self: ContractState,
    snos_output: Span<felt252>,
    layout_bridge_output: Span<felt252>, // ...
) {
    // ...
    let program_info = self.config.program_info.read();
    // @audit The layout bridge program hash is correctly checked at index 2.
    let layout_bridge_program_hash = layout_bridge_output.at(2);
    assert(
        program_info.layout_bridge_program_hash == *layout_bridge_program_hash,
        errors::LAYOUT_BRIDGE_INVALID_PROGRAM_HASH,
    );
    let snos_output_hash = poseidon_hash_span(snos_output);
    // @audit The snos output hash is correctly checked at index 4.
    let snos_output_hash_in_bridge_output = layout_bridge_output.at(4);
    assert(
        snos_output_hash == *snos_output_hash_in_bridge_output, errors::SNOS_INVALID_OUTPUT_HASH,
    );
    let output_hash = poseidon_hash_span(layout_bridge_output);
    // ...
    // @audit The fact is calculated using the bootloader program hash from storage.
    // @audit-issue However, there is no check on the program hash at `layout_bridge_output.at(3)`.
    let fact = poseidon_hash_span(
        array![program_info.bootloader_program_hash, output_hash].span(),
    );
    let verifications = IFactRegistryDispatcher {
        contract_address: self.config.get_facts_registry(),
    }
        .get_all_verifications_for_fact_hash(fact);
    if verifications.len() == 0 {
        core::panic_with_felt252(errors::NO_FACT_REGISTERED)
    }; // ...
}
```

Recommendation(s): It is understood that the element at index `3` of the `layout_bridge_output` span represents a program hash integral to the generation of this output. This program hash should be explicitly validated. Consider adding an assertion to the `update_state(...)` function to check the hash at `layout_bridge_output.at(3)` against a corresponding expected program hash value stored in the contract's configuration (e.g., within `program_info` or a dedicated configuration field if it's a different program hash than the main bootloader). This ensures that the `layout_bridge_output` is consistent with the expected program execution pathway and prevents processing outputs derived from unverified or unexpected programs.

Status: Fixed

Update from the client: This has been addressed in ce5e895b.

In the current implementation, the program hash that should match this value is the bootloader program hash. Since the layout bridge is a verifier written in Cairo, the verification happens on a bootloaded Starknet OS program.

The bootloader program hash is currently not configurable by the operator, and it is defined by SHARP that is bootloading the jobs.

This opens an issue if such bootloader program is changed (which is considered very unlikely at the moment) without prior notice, which could block Piltover. However this can be solved by updating the program info.

6.4 [Info] State can be updated with KZG data availability without verification

File(s): src/snos_output.cairo

Description: The update_state(...) function is responsible for processing the Starknet OS (SNOS) output to verify and apply state transitions. The SNOS output format supports two modes for data availability: on-chain data and KZG (Kate-Zaverucha-Goldberg) data availability. If KZG DA is used, indicated by the use_kzg_da flag, the state diff is not included directly in the output. Instead, commitments to the state diff are provided, which must be verified against KZG proofs for the data blobs.

The deserialize_os_output(...) function correctly identifies when KZG DA is being used by checking the use_kzg_da flag. However, it only consumes the KZG-related data from the input stream without performing any verification.

```
pub fn deserialize_os_output(ref input_iter: SpanIter<felt252>) -> StarknetOsOutput {
    // ...
    let header = read_segment(ref input_iter, HEADER_SIZE);
    let use_kzg_da = header[USE_KZG_DA_OFFSET];
    // ...
    if use_kzg_da.is_non_zero() {
        let kzg_segment = read_segment(ref input_iter, 2);
        let n_blobs: usize = (*kzg_segment.at(KZG_N_BLOBS_OFFSET))
            .try_into()
            .expect('Invalid n_blobs');
        // @audit-issue If `use_kzg_da` is non-zero, the function reads the segment but no verification of the
        ↪ corresponding KZG proofs is performed.
        let _ = read_segment(ref input_iter, 2 * 2 * n_blobs);
    }
    // ...
}
```

The current implementation of the update_state(...) function in Piltover does not support handling the KZG DA case. If a SNOS output is submitted with the use_kzg_da flag set to 1, the function will proceed without verifying the corresponding state_diff, as the logic to handle KZG proofs is absent. This means that a key component of the state transition verification is missing when this data availability scheme is used.

Recommendation(s): Consider implementing a robust mechanism to handle the KZG data availability case. A recommended approach, similar to Starknet's official L1 contract, is to introduce a separate entry point, such as update_state_kzg_da(...), which accepts the necessary KZG proofs as arguments. This function should perform the required cryptographic verification of the proofs against the commitments present in the SNOS output.

Alternatively, if supporting KZG DA is not an immediate priority, consider adding a check to the deserialize_os_output(...) function to revert the transaction if use_kzg_da is non-zero, thereby explicitly disallowing state updates via KZG DA until the verification logic is fully implemented.

Status: Fixed

Update from the client: Since the KZG DA processing requires a totally different handle of the output and KZG proofs, the support has been explicitely removed in a6e5511c where piltover will reject any Starknet OS (SNOS) output with the use_kzg_da flag set to true.

Since no work has been done yet to correctly implement a new entrypoint update_state_kzg_da to accept KZG proofs, the support will be added in a future update of piltover.

6.5 [Info] Subsequent calls to `start_message_cancellation(...)` will overwrite the previous cancellation time

File(s): `src/messaging/component.cairo`

Description: Users of the Piltover system can request to cancel messages sent from Starknet to an Appchain while they are in a pending state. This is a two-step process. First, the user calls the `start_message_cancellation(...)` function, which records that a cancellation has been initiated for a specific message. This makes the message eligible for final cancellation after a predetermined delay (e.g., 5 days), storing the timestamp of this initiation. The second step involves calling `cancel_message(...)` after this delay has passed, which, if the message has not been sealed by then, marks it as cancelled.

However, the `start_message_cancellation(...)` function does not check if a cancellation process has already been started for the given message. If `start_message_cancellation(...)` is called again for the same message hash, it will overwrite the existing timestamp in `sn_to_appc_cancellations` with the new current block timestamp.

This means that each subsequent call to `start_message_cancellation(...)` for the same message effectively resets the waiting period required before `cancel_message(...)` can be successfully executed. While it is unlikely that a user would intentionally do this, an accidental or misinformed repeated call could prolong the time until the message can be cancelled. This presents a minor operational inconvenience, as it could delay the user's ability to finalize the cancellation.

```
fn start_message_cancellation(...) -> MessageHash {
    // ...
    // @audit-issue Subsequent calls will overwrite the previous cancellation start time.
    self.sn_to_appc_cancellations.write(message_hash, starknet::get_block_timestamp());
    // ...
    return message_hash;
}
```

Recommendation(s): Consider revisiting the logic of `start_message_cancellation(...)` to prevent the cancellation waiting period from being unintentionally reset by multiple calls for the same message.

Status: Fixed

Update from the client: Addressed in 4ca0a3d0 where a new status `Cancelling` has been added. This new status is useful to clarify the exact status of the message, which also ensures that the `start_message_cancellation` can't be called twice on the same message.

6.6 [Info] The `DataAvailabilityFact` elements are not validated in the `update_state(...)` function

File(s): `src/appchain.cairo`

Description: The `update_state(...)` function in the `appchain` contract is called by whitelisted operators to process state transitions for the Appchain. It takes the StarknetOS program output (`snos_output`) as an input, which includes the complete state diff, meaning the primary data for state recreation is available in calldata on Starknet. The function also accepts `onchain_data_hash` and `onchain_data_size` as input parameters. These two parameters are used to construct a `DataAvailabilityFact` structure. This `DataAvailabilityFact`, in conjunction with the `layout_bridge_output` (another input to the function), is used to create the `state_transition_fact`. This `state_transition_fact` is then emitted in an event for every state update.

The problem in the `update_state(...)` function is that while other inputs like `snos_output` and `layout_bridge_output` undergo several validation checks, the `onchain_data_hash` parameter, which is a crucial component of the `DataAvailabilityFact`, is not validated. An operator can provide any arbitrary `felt252` value for `onchain_data_hash` when calling this function.

This allows a whitelisted operator to supply a potentially misleading `onchain_data_hash`. Consequently, an incorrect `state_transition_fact` would be generated using this unverified hash and emitted as an event. Although the ability to call `update_state(...)` is restricted to a whitelisted group of operators, this lack of validation can lead to the emission of a `state_transition_fact` that inaccurately represents data availability details. This compromises the integrity of the information emitted in the `state_transition_fact`.

```
fn update_state(
    ref self: ContractState,
    snos_output: Span<felt252>,
    layout_bridge_output: Span<felt252>,
    // @audit The onchain_data_hash parameter is an input from the operator.
    onchain_data_hash: felt252,
    onchain_data_size: u256,
) {
    // ...
    // @audit-issue The onchain_data_hash is used here directly as received from the input,
    // without any prior validation within this function.
    let data_availability_fact: DataAvailabilityFact = DataAvailabilityFact {
        onchain_data_hash, onchain_data_size,
    };

    // @audit The state_transition_fact is constructed using the potentially unvalidated
    // onchain_data_hash via the data_availability_fact.
    let state_transition_fact: u256 = encode_fact_with_onchain_data(
        layout_bridge_output, data_availability_fact,
    );
    // ...
    // @audit The potentially incorrect state_transition_fact is emitted.
    self.emit(LogStateTransitionFact { state_transition_fact });
    // ...
}
```

Recommendation(s): Consider implementing validation for the `onchain_data_hash` parameter to ensure it accurately reflects the intended on-chain data source or commitment. Alternatively, if this hash is not deemed essential or its direct validation is complex within this context, evaluate its necessity in the `DataAvailabilityFact` and the emitted `state_transition_fact`.

Status: Fixed

Update from the client: Since the usage of such values is not well defined in the Piltover context, to avoid any operator manipulation but not derive too much from the Starknet implementation in solidity, a proposal in ec5b1bcd enforces those values to be 0.

6.7 [Best Practices] Missing input validation of the StarknetOS output fields

File(s): `src/state/component.cairo`

Description: The `update(...)` function within the `state` component is invoked when a valid state update has been submitted by an operator through the `update_state(...)` function. Its role is to refresh the component's storage with the latest verified Appchain state information, including the block number, block hash, and state root. This stored information is subsequently queried by operators to determine the next state update required.

The current implementation of `update(...)` correctly asserts that the `prev_block_number` from the `program_output` aligns with the stored `block_number`, and that the `initial_root` matches the stored `state_root`. However, when compared to the Solidity implementation in the `Starknet.sol` L1 core contract, two input validation checks are absent before the state is updated:

1. A check to ensure that the `new_block_number` from the `program_output` is strictly greater than the currently stored block number;
2. A check to ensure that the `prev_block_hash` from the `program_output` is identical to the currently stored block hash;

```
fn update(ref self: ComponentState<TContractState>, program_output: StarknetOsOutput) {
    assert(
        self.block_number.read() == program_output.prev_block_number,
        errors::INVALID_BLOCK_NUMBER,
    );
    // @audit-issue Two checks are missing as compared to Starknet.sol implementation:
    // - new block number MUST be greater than previous block number
    // - the previous block hash from program output MUST be the same as the last stored
    //   hash
    self.block_number.write(program_output.new_block_number);
    self.block_hash.write(program_output.new_block_hash);

    assert(
        self.state_root.read() == program_output.initial_root, errors::INVALID_PREVIOUS_ROOT,
    );
    self.state_root.write(program_output.final_root);
}
```

The `program_output` that is passed to the `update_state(...)` function also includes the `full_output` and `os_program_hash` fields which are not validated anywhere in the code.

1. The `full_output` flag can be set to true or false by the orchestrator. If this flag is not consistently handled or validated, it might complicate the process for off-chain components attempting to accurately recreate the Appchain's state;
2. The `os_program_hash` field is expected to be zero if the StarknetOS (SNOS) is run directly, which is the anticipated scenario for the Appchain contract. If an aggregator were used, this field would contain the hash of the SNOS program;

While not strictly necessary, the missing checks could be added for consistency reasons with the reference implementation and to decrease the attack surface area.

Recommendation(s): Consider adding two additional validation steps to the `update(...)` function, similar to those found in the `Starknet.sol` reference implementation.

Additionally, consider introducing validation for the `full_output` flag and the `os_program_hash` field from the `program_output`.

Status: Fixed

Update from the client: This issue has been addresses in 6121c750 where:

1. The previous block hash in the Starknet OS (SNOS) output needs to match the current piltover state;
2. The new block number must be greater than the current piltover state. One exception is for the genesis block, where the special value of `felt252::MAX` can be followed by block number `0` (or more in future SNOS implementations);

No assumptions are made for the block number to be `prev_block_number + 1`. It is currently how SNOS works (processing only one block in an execution), but this will change in the future and the proposed changes will still be compatible. 3. Complementary change has been done in a48e208d to ensure the `full_output` is not supported when set to true.

The `os_program_hash` is for now expected to be 0, which is now enforced in 2007539c. However, this will change in the future once applicative recursion will become available, and the changes will be done accordingly.

6.8 [Best Practices] Unused code

File(s): `src/snos_output.cairo`

Description: The `ContractChanges` struct from `snos_output.cairo` file remains unused in the codebase. Since state diffs are currently not utilized, this struct can be safely removed from the codebase to keep the code clean and readable.

Recommendation(s): Consider removing the unused `ContractChanges` struct.

Status: Fixed

Update from the client: Addressed in fd2c9554.

7 Documentation Evaluation

Software documentation refers to the written or visual information that describes the functionality, architecture, design, and implementation of software. It provides a comprehensive overview of the software system and helps users, developers, and stakeholders understand how the software works, how to use it, and how to maintain it. Software documentation can take different forms, such as user manuals, system manuals, technical specifications, requirements documents, design documents, and code comments. Software documentation is critical in software development, enabling effective communication between developers, testers, users, and other stakeholders. It helps to ensure that everyone involved in the development process has a shared understanding of the software system and its functionality. Moreover, software documentation can improve software maintenance by providing a clear and complete understanding of the software system, making it easier for developers to maintain, modify, and update the software over time. Smart contracts can use various types of software documentation. Some of the most common types include:

- Technical whitepaper: A technical whitepaper is a comprehensive document describing the smart contract's design and technical details. It includes information about the purpose of the contract, its architecture, its components, and how they interact with each other;
- User manual: A user manual is a document that provides information about how to use the smart contract. It includes step-by-step instructions on how to perform various tasks and explains the different features and functionalities of the contract;
- Code documentation: Code documentation is a document that provides details about the code of the smart contract. It includes information about the functions, variables, and classes used in the code, as well as explanations of how they work;
- API documentation: API documentation is a document that provides information about the API (Application Programming Interface) of the smart contract. It includes details about the methods, parameters, and responses that can be used to interact with the contract;
- Testing documentation: Testing documentation is a document that provides information about how the smart contract was tested. It includes details about the test cases that were used, the results of the tests, and any issues that were identified during testing;
- Audit documentation: Audit documentation includes reports, notes, and other materials related to the security audit of the smart contract. This type of documentation is critical in ensuring that the smart contract is secure and free from vulnerabilities.

These types of documentation are essential for smart contract development and maintenance. They help ensure that the contract is properly designed, implemented, and tested, and they provide a reference for developers who need to modify or maintain the contract in the future.

Remarks about Piltover documentation

The `Piltover` team did not provide any written documentation for the smart contracts for the scope of the audit, but instead provided a comprehensive walkthrough of the project in the kick-off call with a detailed explanation of the intended functionalities. Moreover, the team addressed all questions and concerns raised by the Nethermind Security team, providing valuable insights and a comprehensive understanding of the project's technical aspects.

8 Test Suite Evaluation

8.1 Tests Output

```
> scarb test

Collected 37 test(s) from piltover package
Running 31 test(s) from src/
[PASS] piltover::messaging::tests::test_messaging::cancel_message_cancellation_not_allowed_yet (gas: ~380)
[PASS] piltover::config::tests::test_config::config_set_program_info_unauthorized (gas: ~170)
[PASS] piltover::messaging::tests::test_messaging::message_to_starknet_deser (gas: ~1)
[PASS] piltover::config::tests::test_config::config_set_facts_registry_unauthorized (gas: ~170)
[PASS] piltover::config::tests::test_config::config_register_multiple_operators_ok (gas: ~312)
[PASS] piltover::config::tests::test_config::config_set_facts_registry_ok (gas: ~311)
[PASS] piltover::config::tests::test_config::config_unregister_multiple_operators_ok (gas: ~191)
[PASS] piltover::config::tests::test_config::config_register_operator_ok (gas: ~239)
[PASS] piltover::config::tests::test_config::config_unregister_operator_ok (gas: ~179)
[PASS] piltover::messaging::tests::test_messaging::start_cancellation_invalid_nonce (gas: ~169)
[PASS] piltover::state::tests::test_state::genesis_state_update_ok (gas: ~236)
[PASS] piltover::state::tests::test_state::state_update_invalid_block_number (gas: ~296)
[PASS] piltover::config::tests::test_config::config_set_operator_unauthorized (gas: ~171)
[PASS] piltover::messaging::tests::test_messaging::process_messages_to_appchain_no_seal (gas: ~2)
[PASS] piltover::messaging::tests::test_messaging::cancel_message_ok (gas: ~391)
[PASS] piltover::messaging::tests::test_messaging::start_cancellation_no_message_to_cancel (gas: ~169)
[PASS] piltover::state::tests::test_state::state_update_invalid_previous_root (gas: ~297)
[PASS] piltover::messaging::tests::test_messaging::consume_message_from_appchain_invalid_to_consume (gas: ~4)
[PASS] piltover::config::tests::test_config::config_set_program_info_ok (gas: ~513)
[PASS] piltover::messaging::tests::test_messaging::message_to_appchain_deser (gas: ~1)
[PASS] piltover::messaging::tests::test_messaging::start_cancellation_ok (gas: ~313)
[PASS] piltover::messaging::tests::test_messaging::consume_message_from_appchain_ok (gas: ~10)
[PASS] piltover::messaging::tests::test_messaging::cancel_message_no_message_to_cancel (gas: ~169)
[PASS] piltover::messaging::tests::test_messaging::gather_messages_from_output_ok (gas: ~5)
[PASS] piltover::messaging::tests::test_messaging::appchain_to_sn_messages_ok (gas: ~133)
[PASS] piltover::messaging::tests::test_messaging::process_messages_to_appchain_ok (gas: ~204)
[PASS] piltover::state::tests::test_state::state_update_ok (gas: ~300)
[PASS] piltover::messaging::tests::test_messaging::cancel_message_cancellation_not_requested (gas: ~307)
[PASS] piltover::messaging::tests::test_messaging::process_messages_to_starknet_ok (gas: ~133)
[PASS] piltover::messaging::tests::test_messaging::send_message_ok (gas: ~303)
[PASS] piltover::messaging::tests::test_messaging::sn_to_appchain_messages_ok (gas: ~311)
Running 6 test(s) from tests/
[PASS] piltover_tests::test_appchain::snos_output_deser (gas: ~7)
[PASS] piltover_tests::test_appchain::appchain_owner_only (gas: ~234)
[PASS] piltover_tests::test_appchain::appchain_owner_ok (gas: ~496)
[PASS] piltover_tests::test_appchain::constructor_ok (gas: ~231)
[PASS] piltover_tests::test_appchain::two_step_ownership_transfer_ok (gas: ~253)
[PASS] piltover_tests::test_appchain::update_state_ok (gas: ~1118)
Tests: 37 passed, 0 failed, 0 skipped, 0 ignored, 0 filtered out
```

Remarks about the Piltover Test Suite

The test suite for the Piltover system currently covers its basic operational flows. To further strengthen the test coverage, it is recommended to expand existing tests with a wider variety of scenarios. This should include more thorough testing of unhappy paths and edge cases involving incorrect or unexpected input variations to ensure robust error handling and system resilience under diverse conditions.

Building on this, the main suggestion to significantly enhancing the testing strategy is the development of a comprehensive end-to-end test suite. Ideally, this suite would enable local simulation of the complete state update and messaging lifecycle. This includes the off-chain `Orchestrator`'s operations—from data gathering and proof generation (simulating StarknetOS and Layout Bridge outputs) through to fact submission—and its interactions with on-chain components like the `Appchain` contract and a mock or locally deployed Herodotus Fact Registry.

Such an end-to-end testing environment would offer considerable benefits. It would make it significantly easier to reason about the system's behavior as a whole, particularly the interplay between the off-chain and on-chain parts. Furthermore, it would serve as a valuable framework for testing new features or proof-of-concept ideas efficiently, thereby supporting future development and verification efforts, including subsequent audits.

9 About Nethermind

Nethermind is a Blockchain Research and Software Engineering company. Our work touches every part of the web3 ecosystem - from layer 1 and layer 2 engineering, cryptography research, and security to application-layer protocol development. We offer strategic support to our institutional and enterprise partners across the blockchain, digital assets, and DeFi sectors, guiding them through all stages of the research and development process, from initial concepts to successful implementation.

We offer security audits of projects built on EVM-compatible chains and Starknet. We are active builders of the Starknet ecosystem, delivering a node implementation, a block explorer, a Solidity-to-Cairo transpiler, and formal verification tooling. Nethermind also provides strategic support to our institutional and enterprise partners in blockchain, digital assets, and decentralized finance (DeFi). In the next paragraphs, we introduce the company in more detail.

Blockchain Security: At Nethermind, we believe security is vital to the health and longevity of the entire Web3 ecosystem. We provide security services related to Smart Contract Audits, Formal Verification, and Real-Time Monitoring. Our Security Team comprises blockchain security experts in each field, often collaborating to produce comprehensive and robust security solutions. The team has a strong academic background, can apply state-of-the-art techniques, and is experienced in analyzing cutting-edge Solidity and Cairo smart contracts, such as ArgentX and StarkGate (the bridge connecting Ethereum and StarkNet). Most team members hold a Ph.D. degree and actively participate in the research community, accounting for 240+ articles published and 1,450+ citations in Google Scholar. The security team adopts customer-oriented and interactive processes where clients are involved in all stages of the work.

Blockchain Core Development: Our core engineering team, consisting of over 20 developers, maintains, improves, and upgrades our flagship product - the Nethermind Ethereum Execution Client. The client has been successfully operating for several years, supporting both the Ethereum Mainnet and its testnets, and now accounts for nearly a quarter of all synced Mainnet nodes. Our unwavering commitment to Ethereum's growth and stability extends to sidechains and layer 2 solutions. Notably, we were the sole execution layer client to facilitate Gnosis Chain's Merge, transitioning from Aura to Proof of Stake (PoS), and we are actively developing a full-node client to bolster Starknet's decentralization efforts. Our core team equips partners with tools for seamless node set-up, using generated docker-compose scripts tailored to their chosen execution client and preferred configurations for various network types.

DevOps and Infrastructure Management: Our infrastructure team ensures our partners' systems operate securely, reliably, and efficiently. We provide infrastructure design, deployment, monitoring, maintenance, and troubleshooting support, allowing you to focus on your core business operations. Boasting extensive expertise in Blockchain as a Service, private blockchain implementations, and node management, our infrastructure and DevOps engineers are proficient with major cloud solution providers and can host applications in-house or on clients' premises. Our global in-house SRE teams offer 24/7 monitoring and alerts for both infrastructure and application levels. We manage over 5,000 public and private validators and maintain nodes on major public blockchains such as Polygon, Gnosis, Solana, Cosmos, Near, Avalanche, Polkadot, Aptos, and StarkWare L2. Sedge is an open-source tool developed by our infrastructure experts, designed to simplify the complex process of setting up a proof-of-stake (PoS) network or chain validator. Sedge generates docker-compose scripts for the entire validator set-up based on the chosen client, making the process easier and quicker while following best practices to avoid downtime and being slashed.

Cryptography Research: At Nethermind, our Cryptography Research team is dedicated to continuous internal research while fostering close collaboration with external partners. The team has expertise across a wide range of domains, including cryptography protocols, consensus design, decentralized identity, verifiable credentials, Sybil resistance, oracles, and credentials, distributed validator technology (DVT), and Zero-knowledge proofs. This diverse skill set, combined with strong collaboration between our engineering teams, enables us to deliver cutting-edge solutions to our partners and clients.

Smart Contract Development & DeFi Research: Our smart contract development and DeFi research team comprises 40+ world-class engineers who collaborate closely with partners to identify needs and work on value-adding projects. The team specializes in Solidity and Cairo development, architecture design, and DeFi solutions, including DEXs, AMMs, structured products, derivatives, and money market protocols, as well as ERC20, 721, and 1155 token design. Our research and data analytics focuses on three key areas: technical due diligence, market research, and DeFi research. Utilizing a data-driven approach, we offer in-depth insights and outlooks on various industry themes.

Our suite of L2 tooling: Warp is Starknet's approach to EVM compatibility. It allows developers to take their Solidity smart contracts and transpile them to Cairo, Starknet's smart contract language. In the short time since its inception, the project has accomplished many achievements, including successfully transpiling Uniswap v3 onto Starknet using Warp.

- **Voyager** is a user-friendly Starknet block explorer that offers comprehensive insights into the Starknet network. With its intuitive interface and powerful features, Voyager allows users to easily search for and examine transactions, addresses, and contract details. As an essential tool for navigating the Starknet ecosystem, Voyager is the go-to solution for users seeking in-depth information and analysis;

- **Horus** is an open-source formal verification tool for StarkNet smart contracts. It simplifies the process of formally verifying Starknet smart contracts, allowing developers to express various assertions about the behavior of their code using a simple assertion language;

- **Juno** is a full-node client implementation for Starknet, drawing on the expertise gained from developing the Nethermind Client. Written in Golang and open-sourced from the outset, Juno verifies the validity of the data received from Starknet by comparing it to proofs retrieved from Ethereum, thus maintaining the integrity and security of the entire ecosystem.

Learn more about us at nethermind.io.

General Advisory to Clients

As auditors, we recommend that any changes or updates made to the audited codebase undergo a re-audit or security review to address potential vulnerabilities or risks introduced by the modifications. By conducting a re-audit or security review of the modified codebase, you can significantly enhance the overall security of your system and reduce the likelihood of exploitation. However, we do not possess the authority or right to impose obligations or restrictions on our clients regarding codebase updates, modifications, or subsequent audits. Accordingly, the decision to seek a re-audit or security review lies solely with you.

Disclaimer