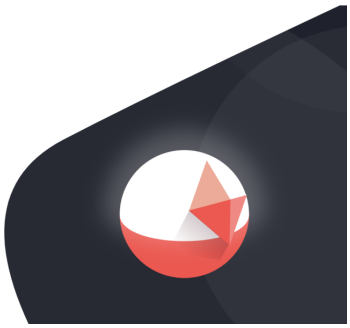
OPTIMISM

Smart Contracts Audit

OpenZeppelin | security



The Optimism team is building the OVM, a fully-featured, EVM-compliant execution environment designed for Layer 2 systems. Starting on March 15th, 2021, we have audited Optimism's code base during 7 weeks with 3 auditors.

The engagement involved auditing two different versions of the Solidity smart contracts:

- The first audited commit was `18e128343731b9bde23812ce932e24d81440b6b7`. We worked with this commit for the first 4 weeks.
- The second audited commit was `a935e276f5620b40802b52721e3474232e458f72`. We worked with this commit for the last 3 weeks.

The Solidity files in scope were those in the `contracts/optimistic-ethereum/OVM/` and `contracts/optimistic-ethereum/libraries/` folders, except the `OVM_BondManager.sol`, `OVM_SafetyChecker.sol`, `ERC1820Registry.sol` and `OVM_DeployerWhitelist.sol` files. It should be noted that while the `Lib_RingBuffer.sol` file was originally included in the audit's scope, after feedback during the audit the file has been deprecated and is expected to change in the short term - we have therefore not conducted a full assessment of the security of this particular library. Moreover, the specified commits contain production code that has been temporarily commented out. Our analysis assumes it will be restored.

All other components of the stack (such as compilers, off-chain services, Layer 2 nodes, or bytecode checkers) were left completely out scope from the beginning of the audit. We have assumed any out-of-scope component behaves as intended and documented by the Optimism team.

Furthermore, the Optimism team independently identified a number of issues in the code base that they shared with us during the audit. For completeness, these are included in an informational note titled **"[N01] Additional issues"**.

Summary

During the audit we were able to uncover issues of varied nature. The most important relate to critical security flaws in the fraud proof verification process, which is the fundamental mechanic ensuring security of the Layer 2 system. We also detected particularly interesting issues in cross-domain deposits and withdrawals of tokens, mishandling of transaction fees, mismatches in the treatment of sequenced and queued transactions, as well as potential abuses of the reward dynamics during fraud proof contribution. Additionally, while not strictly pertaining to security-related issues, this report includes a significant number of recommendations aimed at improving the overall quality of the system.

In terms of specification and documentation, even though there have been notable advances, we still find that there is room for improvements. Gas accounting, upgradeability of core modules, genesis process, progressive decentralization roadmap, and interactions between Layer 1 contracts and off-chain services (such as the Sequencer), stand out as significant attention points in this regard. We acknowledge that the system is still under development, which can render documentation efforts pointless. Still, we expect that as the system matures and its fundamental mechanics are settled, the Optimism team will make efforts to produce a comprehensive baseline specification of their layer 2 solution that can serve as the bedrock upon which future versions of the system build.

Regarding overall health and maturity, we found the code to be readable and well-structured, with enough separation of concerns and modularity to favor long-term maintenance and sustainability. Having audited an earlier version of the system in November 2020, this time we found a more robust code base with simpler and more straightforward implementations. The remarkable difference in number and severity of issues identified between audits is a reflection of how much the code base has matured in the last months. We expect the system to continue in this path as it further evolves, incorporating additional feedback from security-minded professionals, development partners and users alike.

As a final remark, we must highlight that given the sandboxed environment allows for arbitrary code execution of Layer 2 transactions, the number of possible interactions cannot be audited to exhaustion. We therefore highly advise:

- Following best practices of secure software development, thorough test-driven development, and mandatory peer-reviews.
- Further promoting a public bug bounty program to engage independent security researchers from the community in uncovering further misbehaviors in the system as the code base evolves.
- Continuing with beta testing phases until several projects have been onboarded to the Layer 2 system, and their dynamics have become battle-tested, in particular with respect to fraud proof

verification in Layer 1. In this regard, we value the slow progressive decentralization approach taken by the Optimism team.

System overview

Documentation about the system main components can be found in the official documentation and research articles, as well as in our original audit report. On top of the core components audited in the early version of the system, in this audit we also included:

- Bridge contracts, which essentially allow for message passing between Layer 1 and 2.
- Predeploy contracts, which provide essential utilities in Layer 2 (such as a tokenized versioned of ETH, or decompression of sequenced calldata).

Privileged roles

- The owner of the `Lib_AddressManager` contract can arbitrarily add, delete and modify the addresses stored. It is therefore entitled to impersonate or change the logic of critical components of the system at will.
- The Sequencer is a single, semi-trusted, off-chain service that is expect to process, order and append batches of transactions to the Canonical Transaction chain.
- There are accounts that can propose state roots in the State Commitment chain. These accounts should have deposited a bond to be granted the role, and are expected to be penalized if they misbehave.
- As discussed further in the report (see issue **"[M03] Initial state root cannot be challenged"**), the entity that provides the first state root to the State Commitment Chain can decide on the OVM's genesis state.

Update

All serious issues have been fixed or acknowledged by the Optimism team. The code has been migrated to a new repository with a different commit history. Our review disregards all changes introduced by the migration, other pull requests, or unrelated changes within the reviewed pull requests.

Below, our findings in order of importance.

Critical severity

[C01] Possible state manipulation after execution of transactions with invalid gas limit

The `run` function of the `OVM_ExecutionManager` contract is executed during a fraud proof to move the associated State Transitioner from pre-execution to post-execution phase. Transactions with an invalid gas limit can still be run in a fraud proof, but they are expected to ultimately result in a no-op. In other words, the pre- and post-state of a transaction with an invalid gas limit should be the same.

Right before finishing execution of a regular transaction (that is, one that does not revert nor return early), the `run` function resets to zero the reference to the State Manager stored in the `ovmStateManager` state variable. However, this reference is not reset after finishing execution early when the transaction's gas limit is not valid. As a result, the Execution Manager will still be able to execute calls to the associated State Manager, and in turn the State Manager will consider the caller Execution Manager as correctly authenticated.

The described behavior allows for anyone to call sensitive functions of the Execution Manager contract right after the execution of `run`, which could result in arbitrary state modifications in the State Manager contract. As a consequence, it would not be possible to successfully finish the fraud proof in the State Transitioner contract. For example, a malicious actor could call the `ovmCREATEEOA` function of the `OVM_ExecutionManager` contract, creating a new account in state, which would in turn increment the total number of uncommitted accounts in the State Manager. This would effectively prevent completing the transition in the `OVM_StateTransitioner` contract.

Consider clearing the `ovmStateManager` state variable after execution of transactions with invalid gas limits.

***Update**: Fixed in pull request #366 of the archived `ethereum-optimism` repository.*

[C02] Partially shared keys with EXTENSION nodes mishandled

Inside the `_walkNodePath` function of the `Lib_MerkleTrie` library, when an `EXTENSION` node is encountered which shares some, but not all of its key with `keyRemainder`, the walk will move on to the node which the `EXTENSION` node points to, but will only increment the key index by the `sharedNibbleLength`.

More specifically, when the `sharedNibbleLength` is not 0, it will be assumed that all nibbles are shared with the `EXTENSION` node. The walk will then move to the node which the node links to, while the key increment will be set to `sharedNibbleLength`. This will result in an incorrect `currentKeyIndex` on the next loop iteration. The `_walkNodePath` function will assume it has reached the node which the `EXTENSION` node points to, while the `currentKeyIndex` will correspond to a path in the middle of the `EXTENSION` node.

This means that the Merkle Trie may incorrectly identify some elements and there may be multiple keys that map to the same element. During fraud proof execution the described flaw could cause the `OVM_StateTransitioner` contract to incorrectly update storage or account elements, which could lead to a security vulnerability where invalid fraud proofs would succeed due to incorrect updates in trie roots.

Since the `_walkNodePath` function should identify the nearest sibling to the key which is being "walked" to, consider modifying the `_walkNodePath` function so that it breaks out of the loop and returns when a non-fully matching `EXTENSION` key is found.

***Update**: Fixed in pull request #747.*

[C03] Unbounded nuisance gas

When a transaction is executed, its nuisance gas budget is limited to the transaction gas limit. Additionally, the nuisance gas is limited in every call frame to the gas provided for that call. However, it is not limited by the available nuisance gas before the call.

As a result, if the remaining nuisance gas budget is below the remaining transaction gas before an external call, the nuisance gas budget for the call frame will be incorrectly increased, and the call will be able to consume more nuisance gas than it should be allowed. In this scenario, the overall nuisance budget calculation performed after the call will negative overflow. In practice, this means there is no limit to the amount of nuisance gas that can be used in a transaction, as long as each call frame restricts its nuisance gas usage to its own regular gas limit.

Consider ensuring the nuisance gas budget of each call frame cannot exceed the overall budget.

***Update**: Fixed in pull request #1366.*

[C04] Valid L1-to-L2 queue transactions may be proven fraudulent spoofing queue origin

When initializing a fraud proof via the `initializeFraudVerification` function of the `OVM_FraudVerifier` contract, the relevant transaction is provided so it can be verified to exist within the Canonical Transaction Chain. However, when verifying a transaction that was added through the L1-to-L2 queue (that is, via the `enqueue` function of the `OVM_CanonicalTransactionChain` contract), the `l1QueueOrigin` field of the transaction is not validated.

As a result, anyone can provide a transaction maliciously setting its `l1QueueOrigin` field to `SEQUENCER_QUEUE` (instead of `L1TOL2_QUEUE`) when disputing a transaction that originated in the L1-to-L2 queue. The gas used by this transaction will be attributed to the wrong queue and any logic that relies on the `ovmL1QUEUEORIGIN` opcode may evaluate incorrectly. Naturally, this will produce a different final state, thus allowing the fraud proof to succeed even if the original transition was valid.

Consider validating the `l1QueueOrigin` field of the provided transaction when initializing a fraud proof.

***Update**: Fixed in pull request #1155.*

High severity

[H01] Valid transactions cannot be enqueued

Transactions to be appended to the Canonical Transaction Chain (CTC) can come from two sources: the L1 queue and the Sequencer. When transactions are enqueued in the `OVM_CanonicalTransactionChain` contract via its public `enqueue` function, they are explicitly limited in size to `MAX_ROLLUP_TX_SIZE` (10000 bytes). However, transactions coming from the Sequencer do not follow the same restriction - they can actually be larger than `MAX_ROLLUP_TX_SIZE`. As a result, it might be impossible to enqueue transactions in L1 that could be effectively included via the Sequencer.

To avoid censorship by the Sequencer, it should be possible for any valid sequenced transaction to instead be enqueued in L1. Therefore, consider enforcing an upper bound of `MAX_ROLLUP_TX_SIZE` in the size of transactions that go through the Sequencer.

***Update**: Fixed in pull request #361 of the archived `ethereum-optimism` repository.*

[H02] Unhandled transfer failures

The token bridge contracts synchronize deposits and withdrawals across the two domains. In particular, whenever tokens are locked on layer 1, the gateway initiates a cross-domain message to mint equivalent tokens on layer 2. Similarly, when tokens are burned on layer 2, the token initiates a cross-domain message to release the funds on layer 1. However, the layer 1 ERC20 gateway does not check the return value of the deposit or withdrawal transfers. This breaks the synchronization for ERC20 contracts that do not revert on failure, since failed deposits on layer 1 will be incorrectly credited on layer 2 and burned tokens on layer 2 may not be released on layer 1.

Consider checking the return value on all token transfers and revert on failure.

***Update**: Fixed in pull request #988.*

[H03] Relayers may not receive transaction fees

Transactions that go through the `execute` function of an instance of the `OVM_ECDSAContractAccount` contract are expected to pay transaction fees to relayers. The function assumes that whoever called it is a relayer, and simply transfers the fee, paid in ovmETH.

However, there are two common cases in which the `execute` function can be called, and in neither of them the fee appears to be correctly paid to relayer accounts.

- For sequenced transactions, their entrypoint is set to the address of the `OVM_ProxySequencerEntrypoint` predeployed contract. Ultimately, it is this proxy who calls the

`execute` function of any `OVM_ECDSAContractAccount` contract. As a result, when the `execute` function queries the `ovmCALLER`, the address returned will be the address of the `OVM_ProxySequencerEntrypoint` contract, and fees will be sent to it. It is worth noting that neither this contract nor its associated implementation `OVM_SequencerEntrypoint` have any kind of functionality to handle the received fees.

- For queued transactions, their entrypoint is set by whoever enqueues the transaction in the Canonical Transaction Chain. If this entrypoint is set to an instance of the `OVM_ECDSAContractAccount` contract, when the transaction is run and the `execute` function is called, the internal call to `ovmCALLER`, will simply return the default address for `ovmCALLER`, which is determined by the `DEFAULT_ADDRESS` constant address of the `OVM_ExecutionManager` contract. As a result, the fees will be sent to this address.

Consider ensuring that when transaction fees are paid from instances of the `OVM_ECDSAContractAccount` contract, fees are correctly transferred to the expected relayer addresses.

Update: *Fixed in pull request #1029. Fees are now transferred to a designated Sequencer Fee Wallet.*

[H04] Irrelevant proof contributions are accepted

The `contributesToFraudProof` modifier declared in the `Abs_FraudContributor` contract is used to reward users for participating in proving fraud.

There are several functions marked with this modifier to assign rewards to users, regardless of whether they are making meaningful proof contributions. Some examples of irrelevant contributions that would grant rewards include:

- contributing irrelevant contract state before a fraud proof.
- contributing irrelevant storage slots before a fraud proof.
- attempting to initialize a fraud proof that has already been initialized. This is possible because if the fraud proof has been initialized, executing the `initializeFraudVerification` function of the `OVM_FraudVerifier` contract will still finish execution rather than reverting.

It should be noted that the `OVM_BondManager` contract is out this audit's scope. Nevertheless, consider changing the `contributesToFraudProof` modifier or the logic within the `OVM_BondManager` contract to prevent abuse of rewards during fraud proofs.

Update: *Acknowledged. The Optimism team decided not to prioritize this issue because it does not apply to the current release, since the bond manager is disabled.*

[H05] Repeatedly exceeding nuisance gas limit

Transactions are provided a nuisance gas budget to limit the amount of overhead that could be required in the pre-execution and post-execution phase of a fraud proof. However, this does not prevent

transactions from breaching the limit, it merely detects when they do so. In effect, they are able to exceed their budget by one operation.

Moreover, the nuisance gas provided to each call frame is limited to the gas provided for that call, which also limits how much of the transaction's nuisance gas it can consume. Therefore, each individual call frame can exceed its budget by one operation, but the transaction will only be charged for the specified budget. Consequently, if a transaction consisted entirely of call frames with minimal gas that maximally exceed their nuisance gas budget, the whole transaction could create significantly more overhead than its budget would suggest. The only limit to this attack is the number of cheap call frames that can fit in a transaction.

The operation that creates the most overhead is interacting with a large contract, since this may require deploying that contract to the EVM during a fraud proof. The Optimism team have indicated they intend to introduce a minimum nuisance gas budget in each call frame that covers the cost of deploying the largest possible EVM contract. While this would mitigate the vulnerability, it should be noted that the call frame could still consume this nuisance gas on other operations before interacting with a large contract, so it would still be possible to consume twice as much nuisance gas as the transaction budget would suggest. Consider documenting this behavior in the function comments.

***Update**: Fixed in pull request #1366. Transactions that could breach their nuisance gas limit are now reverted pre-emptively.*

Medium severity

[M01] Potential mismatch in allowed gas limit for sequenced and queued transactions

Both enqueued and sequenced transactions are expected to be limited in the amount of gas they can consume. For enqueued transactions the `enqueue` function of the `OVM_CanonicalTransactionChain` contract ensures that their `gasLimit` does not exceed the `maxTransactionGasLimit` (a parameter of the Canonical Transaction Chain set during construction by the contract's deployer). Conversely, for sequenced transactions the expected `gasLimit` is only validated during verification of a sequenced transaction and actually retrieved from the Execution Manager. Since there is no logic programmatically enforcing that the gas limit retrieved from the Execution Manager matches the gas limit enforced by the Canonical Transaction Chain, there is room for the `gasLimit` of sequenced and enqueued transactions to be capped differently.

While the described behavior would introduce an unexpected difference between in how enqueued and sequenced transactions are treated in the OVM, it would only occur due to a misconfiguration of the system by its deployers or a corrupt upgrade.

Consider defining a single source of truth that dictates the gas limit for both sequenced and enqueued transactions. Alternatively, consider introducing off-chain validations to ensure these values always match, reverting new configurations or upgrades otherwise.

Update: *Acknowledged. The Optimism team intends to address this in the future.*

[M02] Pre-state root and transaction may not uniquely identify transitions

The `OVM_FraudVerifier` contract identifies State Transitioner contracts using the pre-state root and the transaction hash. The same function is used by the `OVM_BondManager` contract. Note that fraud proofs are intended to remove the post-state root (and subsequent state roots) but they are specified by the combination of pre-state root and transaction hash. However, given the possibility of repeated transactions and state roots, this approach may not uniquely identify a particular transition.

This means that a fraud proof can apply equally to all transitions that use the same pre-state root and transaction hash. However, only one pre-state root can be specified during finalization of the proof. As a result, legitimate fraud proofs that attempt to remove the first invalid state root can be maliciously finalized on a later state root, forcing the original fraud proof to be restarted. This can be achieved by tracking the progress of active fraud proofs or by front-running calls to the `finalizeFraudVerification` function, and could prevent the first invalid state root from being removed indefinitely.

It should be noted that this attack is possible because State Transitioner contracts are removed once they are finalized. The code base includes a comment suggesting that they may be retained in future versions, which would allow the same State Transitioner to be reused for every matching transition. Yet it is worth noting that this may require a redesign of how this function interacts with the Bond Manager. An alternative approach can be based on identifying State Transitioner contracts with the unused `stateTransitionIndex` variable as well to avoid collisions. Interestingly, since it is possible for a state root to be removed from the State Commitment Chain and then be reintroduced at the same location (possibly with a different history and batch structure), this would not necessarily uniquely identify a transition, but in such a case, reusing the State Transitioner contract would be appropriate.

Update: *Acknowledged. This is not exploitable while the Sequencer is assumed to be trusted and the public transaction queue mechanism is disabled. The Optimism team intends to address this in the future.*

[M03] Initial state root cannot be challenged

The `initializeFraudVerification` function of the `OVM_FraudVerifier` contract intends to ensure that the provided pre-state root and transaction correspond to each other. In other words, that the referenced transaction was executed against the provided pre-state. This is implemented in this `require` statement, where the offset suggests that the element at index N in the State Commitment Chain is the post-state root of the transaction at index N in the Canonical Transaction Chain. This is consistent with the fact that the size of the State Commitment Chain is bounded by the total number of transactions in the Canonical Transaction Chain.

However, since fraud proofs require the pre-state to exist in the State Commitment Chain, it is impossible to prove fraud against the first state root in the State Commitment Chain. As a result, the

first transaction in the Canonical Transaction Chain can be considered meaningless, and the first state root in the State Commitment Chain will remain unchallenged. This effectively introduces a remarkable trust assumption, where the entity that provides the first state root can decide on the OVM's genesis state.

Moreover, since state roots are deleted in batches instead of individually, if the genesis state root shares the same batch with other state roots, and one of them is successfully proven fraudulent, the entire batch of state roots (including the genesis state root) will be removed, and therefore the next state root to be appended to the State Commitment Chain will become the new "genesis" state.

Consider thoroughly documenting the deployment procedure, including the fact that the first transaction is unused, and how the Optimism team intends to ensure the first state root will be the intended genesis state. Alternatively, consider introducing a mechanism to challenge the first post-state root against a known genesis state.

***Update**: Acknowledged, but won't fix. Optimism's statement for this issue:*

> The initial state root is analogous to Ethereum's genesis block. It cannot be the result of a fraudulent transaction, users of an Optimistic Ethereum deployment must accept the initial state root in the same way that they accept the state transition rules for any blockchain.

[M04] Sequencer entrypoint contracts ignore success flags and returned data

The `fallback` functions of the `OVM_ProxySequencerEntrypoint` and `OVM_SequencerEntrypoint` contracts respectively execute `ovmDELEGATECALL` and `ovmCALL` calls, yet they fail to handle the success flag returned by these calls, as well as any data returned. As a consequence, calls to these `fallback` functions could fail silently, and might be erroneously seen as successful by callers.

To avoid unexpected errors, consider handling success flags and return data for these calls. It should be noted that fallback function return values were introduced in solidity 0.7.6, so they will not be supported by the full range of target compilers.

***Update**: Fixed for the `OVM_SequencerEntrypoint` case in pull request #603. The `OVM_ProxySequencerEntrypoint` contract was removed in pull request #549.*

[M05] Nuisance gas left is not reduced to zero when operation exceeds budget

The `_useNuisanceGas` function of the `OVM_ExecutionManager` contract is intended to reduce a certain amount of nuisance gas during transaction execution. When the amount of nuisance gas required by the operation exceeds the nuisance gas left, the function reverts the call with flag `EXCEEDS_NUISANCE_GAS`. However, this flag is not taken into account when accounting for the call's nuisance gas consumption within the `_handleExternalMessage` function. Therefore, the nuisance gas

left for the transaction is never reduced to zero, allowing subsequent operations to continue consuming nuisance gas.

For correctness, consider reducing to zero the amount of nuisance gas left whenever a call raises the `EXCEEDS_NUISANCE_GAS` flag.

***Update**: Fixed in pull request #1366. Transactions revert pre-emptively so they will not exceed their nuisance gas limit, but the nuisance gas budget is still consumed.*

Low severity

[L01] Appending transactions to the Canonical Transaction Chain in specific blocks might unexpectedly fail

Enqueued transactions cannot be included by non-Sequencer accounts during the force inclusion period. If a user attempts to do so calling the `appendQueueBatch` function of the `OVM_CanonicalTransactionChain` contract, then the call would revert at this `require` statement. Once the force inclusion period finishes for the next enqueued transaction to be appended, then the Sequencer just cannot ignore it, and therefore the queue transaction *must* be included. This behavior is enforced by this `require` statement executed during a call to the `appendSequencerBatch` function.

However, the `require` statements referenced above enforce strict inequalities, failing to consider the case where the current block's timestamp is equal to the sum of the next queue element's timestamp and the force inclusion period. In this scenario, the Sequencer would be prevented from adding transactions (since it should first append the transaction at the queue's front), but it would also be impossible to add the enqueued transaction via the `appendQueueBatch` function. As a result, attempts to append transactions to the Canonical Transaction Chain in this scenario would unexpectedly fail. It should be noted that the issue would be automatically resolved by waiting for the next block.

Consider modifying one of the two strict inequalities referenced above to ensure enqueued transactions can *always* be appended to the Canonical Transaction Chain, either by the Sequencer or via the `appendQueueBatch` function.

[L02] Inaccessible code when retrieving Merkle roots

There is an inaccessible `if` block in the `getMerkleRoot` function of the `Lib_MerkleTree` library. This is due to the fact that the `require` statement above it reverts in the case that the caller-provided `_elements` array has no elements.

Because inside the `if` block the function is returning the first element of the array, we assume that the original intended behavior was to validate that the `_elements` array has a single element. Consider modifying the condition evaluated to reflect this.

***Update**: This issue was identified in the first audited commit. It is fixed in the latest audited commit.*

[L03] Lack of input validations

In the interest of predictability, some functions could benefit from more stringent input validations.

- The `init` function of the `Abs_L2DepositedToken` abstract contract does not ensure that the passed token gateway address is non-zero. If it is called with a zero address (before the gateway address in state is set to a non-zero value), it will incorrectly emit an `Initialized` event.
- The `getMerkleRoot` function of the `Lib_MerkleTree` library provides 16 default values, which implicitly limits the depth of unbalanced trees to 16. Balanced trees, on the other hand, have no restriction. Although this is unlikely to matter in practice, usage assumptions should be documented and validated wherever possible. Consider explicitly bounding the number of elements by 2^{16}.
- According to the RLP specification described in the Appendix B of Ethereum's Yellow Paper, "*Byte arrays containing 2^{64} or more bytes cannot be encoded*". This restriction is not being explicitly enforced by the `writeBytes` function of the `Lib_RLPWriter` library.
- The `_editBranchIndex` function of the `Lib_MerkleTrie` library should explicitly validate that the passed index is lower than the `TREE_RADIX` constant to avoid misusage.
- The `_getNodePath` function of the `Lib_MerkleTrie` library should explicitly validate that the passed node is a leaf or extension node to avoid misuse.

[L04] Merkle tree elements are overwritten

The `getMerkleRoot` function of the `Lib_MerkleTree` library accepts an array of elements and computes the corresponding Merkle root. In the process, it unexpectedly overwrites up to half of the elements, thereby corrupting the original array. The current code base has one instance, within the `appendStateBatch` function of the `OVM_StateCommitmentChain` contract, where the array is used after it is passed to the `getMerkleRoot` function, but fortunately it only reads the length, which is unchanged. This function calculates the Merkle root of its `_batch` parameter, which means that the caller may attempt to reuse the (now corrupted) array.

Consider including warning documentation on the `getMerkleRoot` and `appendStateBatch` functions stating that the input may be modified.

***Update**: This issue was identified in the first audited commit. It was fixed in the latest audited commit by adding relevant documentation.*

[L05] Misleading and / or erroneous docstrings and comments

In the `OVM_CanonicalTransactionChain` contract:

-

The comment in line 320 should say "the real queue index" instead of "the real queue length".

- The `@return` tag of the `verifyTransaction` function states that the function returns `false` if the transaction does not exist in the Canonical Transaction Chain. However, the function will revert in such scenario.
- The `@return` tags of the `_verifySequencerTransaction` and `_verifyQueueTransaction` functions state that they return `false` upon failure. Yet in such scenario they revert.

In the `OVM_ExecutionManager` contract:

- The `@return` tag of the `ovmL1QUEUEORIGIN` function states that an address is returned, yet the actual returned value is the element of an `enum`.
- The inline comment in line 857 states that the nonce is updated even if contract creation fails, yet that is incorrect. When the contract creation fails with a revert due to the deployer not being allowed, the account's nonce is not updated (see note **"[N10] Contract creation can revert upon failure"** for additional details).
- An inline comment in line 1039 states that the revert flag "*`EXCEEDS_NUISANCE_GAS` explicitly reduces the remaining nuisance gas for this message to zero*". However, as can be observed in the related `_useNuisanceGas` function where the flag is raised, the remaining nuisance gas of the message is not set to zero (as described in issue **"[M05] Nuisance gas left is not reduced to zero when operation exceeds budget"**)
- An inline comment in line 1358 mentions "loading" an account but is referring to changing an account.

In the `Lib_Bytes32Utils` library:

- The `@return` tag of the `removeLeadingZeros` function specifies that the returned value is `bytes32`, while it actually returns a `bytes` type.

In the `OVM_L1ERC20Gateway` contract:

- lines 63 and 64 imply that ETH is being deposited, when actually an ERC20 token is being deposited.

In the `Abs_L1TokenGateway` contract:

- The comment on line 77 describes a withdrawal operation instead of a deposit.
- The comment on line 129 says "withdrawal" instead of "deposit".
- The comment on line 188 says the function will fail if the L2 withdrawal was not finalized, but that logic is not included within the function.

In the `Abs_L2DepositedToken` contract:

-

Docstrings for the contract's constructor should say "L2 Messenger address" instead of "L1 Messenger address".

- Documented parameters `_to` and `_amount` of the `finalizeDeposit` function refer to withdrawals, when they should be referring to deposits.

In the `OVM_L2ToL1MessagePasser` contract:

- Docstrings state that the contract's runtime target is the EVM, while it should say OVM.

In the `OVM_StateManager` contract:

- Docstrings in lines 274 and 292 state that the related functions are only called during `ovmCREATE` or `ovmCREATE2` operations, failing to account that they are also called during `ovmCREATEEOA`.

In the `OVM_StateTransitioner` contract:

- Docstrings for the `getPostStateRoot` function state that the value returned corresponds to the "*state root after execution*". However, if it is called prior to the transaction being applied, the function will return the state root before execution.

In the `Lib_MerkleTrie` library:

- Docstrings for the `_getNodeValue` function should say "Gets the value for a node" instead of "Gets the path for a node".

In the `OVM_ECDSAContractAccount` contract:

- The comment on line 17 indicates that `eth_sign` messages can be parsed, but this functionality has been removed.
- The comment on line 73 appears to be removable as it is similar to the comment on line 79.

[L06] Missing and / or incomplete docstrings

Some contracts and functions in the code base lack documentation or include incomplete descriptions. This hinders reviewers' understanding of the code's intention, which is fundamental to correctly assess not only security, but also correctness. Additionally, docstrings improve readability and ease maintenance. They should explicitly explain the purpose or intention of the functions, the scenarios under which they can fail, the roles allowed to call them, the values returned and the events emitted. Below we list all instances detected during the audit.

- All functions of the `Lib_BytesUtils` library.
- In the `OVM_CanonicalTransactionChain` contract:
- Docstrings for the `_getBatchExtraData` function are missing two `@return` tags.
-

Docstrings for the `_getQueueElement` function are missing the `_queueRef` parameter.

- Docstrings for the `_getSequencerLeafHash` function are missing the `_hashMemory` parameter.
- In the `iOVM_StateCommitmentChain` interface:
- Docstrings for the `verifyStateCommitment` function are missing a `@return` tag.
- In the `OVM_StateCommitmentChain` contract:
- Docstrings for the constructor are missing parameters `_fraudProofWindow` and `_sequencerPublishWindow`.
- In the `OVM_ExecutionManager` contract:
- Docstrings for the constructor are missing the `_gasMeterConfig` and `_globalContext` parameters.
- Docstrings for the `simulateMessage` function are missing the `_ovmStateManager` parameter and the returned values.
- In all predeploy contracts, docstrings could include the address at which each contract will be found in Layer 2.

Consider thoroughly documenting all functions (and their parameters) that are part of the contracts' public API. Functions implementing sensitive functionality, even if not public, should be clearly documented as well. When writing docstrings, consider following the Ethereum Natural Specification Format (NatSpec).

[L07] Undocumented literal values

Throughout the code base, there are several instances of literal values with unexplained meaning. Moreover, some of them are not declared as constant state variables, which further hinders code readability. Literal values in the code base without an explained meaning make the code harder to read, understand and maintain, thus hindering the experience of developers, auditors and external contributors alike. Following we include a list of literal values that should be further documented and explained.

In `OVM_ExecutionManager.sol`:

- Addresses in lines 545, 826 and 908.
- Fixed gas discounts applied in line 391, line 430, line 580, line 614, and line 649.
- Net gas costs applied in line 681 and line 707.
- The value 100 on line 916.

In `OVM_StateTransitioner.sol`:

- The number `100000` in line 341.

In `OVM_L1CrossDomainMessenger.sol`:

- The number `0` in line 246.
- The address `0x4200000000000000000000000000000000000000` in line 254.
- The number `1` in line 270.

In `Lib_MerkleTrie.sol`:

- The number `2` in lines 785 and 811 should be replaced with `LEAF_OR_EXTENSION_NODE_LENGTH`.

In `OVM_CanonicalTransactionChain.sol`:

- The number `15` in line 627 should be replaced with `BATCH_CONTEXT_START_POS`.

Developers should define a constant variable for every literal value used, giving it a clear and self-explanatory name. Additionally, inline comments explaining how they were calculated or why they were chosen are highly recommended. Following Solidity's style guide, constants should be named in `UPPER_CASE_WITH_UNDERSCORES` format, and specific public getters should be defined to read each one of them if appropriate.

[L08] Unspecified behavior of OVM gas refund for revert flags

In the `OVM_ExecutionManager` contract, certain revert flags trigger a refund of OVM gas in the transaction being run. However, there are other flags such as `EXCEEDS_NUISANCE_GAS` and `UNINITIALIZED_ACCESS` which are not taken into account for gas refunds, their expected behavior being unspecified. The `EXCEEDS_NUISANCE_GAS` flag is raised when there is not enough nuisance gas to continue with transaction execution, while the `UNINITIALIZED_ACCESS` flag is raised when the `ovmCALLER` opcode is executed in the transaction's entrypoint.

To better define the behavior of gas refunds in the OVM, consider specifying if and how gas refunds should be applied for the mentioned revert flags.

[L09] Nuisance gas proportional to code size is charged unnecessarily when changing an account

Within the `_checkAccountChange` function of the `OVM_ExecutionManager` contract, nuisance gas is charged proportional to the code size of the account. Since i) it can be assumed that the code deployed in the pre-execution phase of the fraud proof will not change, and ii) nuisance gas proportional to code

size is already charged when initially loading an account for the first time, it appears unnecessary to charge nuisance gas again the first time the account (but not its code) is changed.

Consider removing the nuisance gas fee associated with contract code size within the `_checkAccountChange` function. Note that the solution to this issue might impact what is described in the informational note **"[N08] Minimum nuisance gas per contract creation is charged twice"**.

[L10] Unnecessary handling of single byte returned data

In the `Lib_SafeExecutionManagerWrapper` library, the internal `_safeExecutionManagerInteraction` function handles a case in which the returned data from a call to the `OVM_ExecutionManager` is a single byte. This code segment appears to be outdated, left over from an earlier version of the system, and it is no longer used.

Additionally, the `ovmEXTCODECOPY` function of the `OVM_ExecutionManager` contract introduces an artificial manipulation to avoid users inadvertently triggering this special case.

Consider removing both code segments to favor simplicity and avoid confusion.

[L11] Incorrect state transitioner index

When deploying a new `OVM_StateTransitioner` contract, the `OVM_FraudVerifier` contract incorrectly passes the index of the state root in its corresponding batch, confusing it with the state root's index in the State Commitment chain. The same mistake is made when emitting the `FraudProofInitialized` and `FraudProofFinalized` events.

Consider replacing these values with the index of the state root in the State Commitment chain.

[L12] Inconsistent and error-prone storage references in proxy contracts

There are three different proxy contracts implemented, all of them following a different approach when handling storage references.

- The `OVM_ProxySequencerEntrypoint` contract stores the implementation and owner addresses in continuous storage slots at positions 0 and 1 (as can be seen in the internal getter and setter functions for these addresses). While this approach is certainly simple, it can be considered fragile and error-prone. In particular, any poorly constructed implementation that does not take into account the storage layout of the the proxy might accidentally cause a storage collision, and overwrite these two sensitive proxy variables. The problem is aggravated by the fact that the two state variables are not explicitly declared (and therefore visible) in the contract's state, but rather low-level `ovmSSTORE` and `ovmSLOAD` operations are performed to handle them.
-

The `OVM_ProxyEOA` contract stores the implementation address in the storage slot dictated by its `IMPLEMENTATION_KEY` constant, currently set to `0xdeaddeaddeaddeaddeaddeaddeaddeaddeaddeaddeaddeaddeaddeaddeaddead`.

- The `Lib_ResolvedDelegateProxy` contract does not follow either of the previous approaches. Instead, it uses private mappings in state to store two addresses corresponding to the `Lib_AddressManager` and implementation contracts, which is said to include a "known flaw" without providing further details.

For a more robust and consistent implementation of proxy contracts, consider always following standardized storage slots for relevant addresses in state. Refer to EIP 1967 for one possible approach. This should allow building common, reusable tooling to facilitate reliable inspection and interaction with all proxy contracts in the system.

[L13] Inconsistent key slicing when computing trie root

The `_getUpdatedTrieRoot` function of the `Lib_MerkleTrie` library is intended to compute a Merkle trie root from a given path to a key-value pair. As it iterates through the path, it consumes the corresponding parts of the key by slicing it depending on the type of each node found. Whenever a Leaf or Extension node is found, the key is reduced in size appropriately (see lines 509 and 513), regardless of whether the previous identified node is the last in the path. However, this is not the case for Branch nodes, where the key is only sliced if the previous identified node is not the last in the path.

Consider consistently slicing the key when computing the root of Merkle tries.

[L14] Branch node modification in Merkle Trie may deviate from specification

The `_editBranchIndex` function of the `Lib_MerkleTrie` library does not RLP-encode values if their length is less than 32 bytes. This behavior does not appear to conform with the available specification for Merkle Trie. Consider either complying with the referenced specification, or alternatively document what specification was used for the implementation of the `Lib_MerkleTrie` library.

[L15] Unnecessary use of assembly

- The `getMerkleRoot` function of the `Lib_MerkleTree` library uses assembly to pack elements before hashing them (see here and here). Consider using the globally available `abi.encodePacked` function instead.
- The `slice` function of the `Lib_BytesUtils` library uses assembly to return a zero-length array. Consider replacing these lines simply returning the empty array `tempBytes`.

[L16] Incorrect parsing of booleans in RLP library

The `readBool` function of the `Lib_RLPReader` library converts an RLP-encoded boolean value into a boolean type. The function returns `false` only if the provided value is `0`. However, in Geth `false` is encoded as `0x80`, not `0x00`. For this encoding of `false`, the `readBool` function would incorrectly revert. The flawed behavior can be reproduced by simply attempting to execute `Lib_RLPReader.readBool(Lib_RLPWriter.writeBool(false))`, which currently triggers a revert with message `Invalid RLP boolean value, must be 0 or 1`.

Consider updating the implementation of the `readBool` function to check for the case that the input is `0x80` and return `false` accordingly. Alternatively, given that this function is never used, consider removing it from the code base.

[L17] Lookup key strings are not centrally defined

Known, legitimate contracts are tracked in the `addresses` mapping of the `Lib_AddressManager` contract. New entries can be added by a privileged address via the `setAddress` function, and the `getAddress` function function acts as a public getter to query the registry providing a string-type key. While this registry is used by several different contracts to get the addresses of registered contracts, the strings used as keys to query the registry are not centrally defined. The identified strings are:

- `"OVM_L2MessageRelayer"`
- `"OVM_L2BatchMessageRelayer"`
- `"OVM_StateCommitmentChain"`
- `"OVM_L2CrossDomainMessenger"`
- `"OVM_CanonicalTransactionChain"`
- `"Proxy__OVM_L1CrossDomainMessenger"`
- `"OVM_L1MessageSender"`
- `"OVM_L1CrossDomainMessenger"`
- `"OVM_L2ToL1MessagePasser"`
- `"OVM_ChainStorageContainer:CTC:batches"`
- `"OVM_ChainStorageContainer:CTC:queue"`
- `"OVM_Sequencer"`
- `"OVM_ExecutionManager"`
-

`"OVM_DecompressionPrecompileAddress"`

- `"OVM_ChainStorageContainer:SCC:batches"`
- `"OVM_BondManager"`
- `"OVM_StateCommitmentChain"`
- `"OVM_CanonicalTransactionChain"`
- `"OVM_FraudVerifier"`
- `"OVM_Proposer"`
- `"OVM_SafetyChecker"`
- `"OVM_StateTransitionerFactory"`

While this issue does not pose an immediate security risk, the approach taken can be considered error-prone and difficult to maintain. Moreover, it is worth noting that the current code base has a number of inconsistencies in how these keys are referenced, as described in **"[N06] Inconsistent name resolution"**.

Consider factoring out all mentioned constant strings to a single library, which can be then imported as needed. This will ease maintenance and make the code more resilient to future changes.

[L18] Lack of allowance front-running mitigation in ERC20 contract

The `UniswapV2ERC20` contract does not include the `increaseAllowance` and `decreaseAllowance` functions, nowadays common in most ERC20 interfaces to help mitigate the allowance frontrunning issue of the ERC20 standard.

While not strictly part of the ERC20 standard, consider including these two functions in the contract's interface.

[L19] Lack of event emissions

- In the `OVM_ProxySequencerEntrypoint` contract, the `upgrade` function does not emit an event after a successful upgrade operation.

Consider emitting events after sensitive changes take place to facilitate tracking and notify off-chain clients following the contracts' activity.

[L20] Deployment risks

The following contracts have a public initializer function:

- `OVM_L1CrossDomainMessenger`
- `Abs_L2DepositedToken`
- `OVM_L1ETHGateway`
- `OVM_DeployerWhitelist`
- `OVM_ProxySequencerEntrypoint`

In all cases, the first account to invoke the initializer is not authenticated and can set sensitive parameters, which leaves them open to potential front-running attacks that could invalidate the contracts. We understand that this is particularly relevant for the token bridge contracts, because the Optimism team intends to provide a factory that programmatically creates the contracts and adds them to a registry, so if a particular token bridge is invalidated, it may not be recoverable.

One reason for this pattern is that contracts may have circular deployment dependencies, which means some contracts have to be deployed before their dependencies. Nevertheless, since contract addresses are created deterministically, it should still be possible to predict all addresses before the deployment, and pass them to the relevant constructors. Consider using this method where appropriate to mitigate the risk of front-running during initial configuration transactions. Alternatively, consider implementing access controls to the initializer functions.

Notes & Additional Information

[N01] Additional issues

During our audit, the Optimism team independently found a number of issues in the code base. We briefly include them below for completeness.

- The `passMessageToL1` function of the `OVM_L2ToL1MessagePasser` contract is intended to be called by the `OVM_L2CrossDomainMessenger`, which validates the message sender and nonce before passing on the message. Although the function has no access controls, only messages sent from the `OVM_L2ToL1MessagePasser` can be relayed on layer 1. However, an attacker can use the L1-to-L2 message path to invoke the `passMessageToL1` function from the `OVM_L2CrossDomainMessenger` with any parameters. This lets them send a message back up to L1 that bypasses the message sender and nonce validations. The Optimism team has indicated that they will remove the `OVM_L2ToL1MessagePasser` contract entirely, and use the `sentMessages` mapping in the `OVM_L2CrossDomainMessenger` instead.
-

The logic in the `_handleContractCreation` function of the `OVM_ExecutionManager` contract allows deploying potentially unsafe code. While it does validate the runtime code deployed, the restriction is enforced after the code is already deployed, without reverting the state changes.

- After running a legitimate fraud proof and reaching the post-execution state, the associated State Manager still considers the Execution Manager as "authenticated". This would allow further modifying state during post-execution.
- An attacker can maliciously modify the context in which a fraud proof is run by first calling the `run` function of the `OVM_ExecutionManager` altering context-related variables (such as the `isStatic` flag), and then re-entering it during execution of a fraud proof.

***Update**: These issues were identified in the first audited commit. They are fixed in the latest audited commit. Note that instead of removing the `OVM_L2ToL1MessagePasser` contract, the first issue was addressed by recognizing and discarding L1-to-L2 cross domain messages directed at the `OVM_L2ToL1MessagePasser` contract.*

[N02] Contracts do not compile with Solidity versions prior to 0.7

Contracts throughout the code base explicitly allow to be compiled with Solidity versions lower than 0.8 and greater than 0.5, by setting its `pragma` statement to `pragma solidity >0.5.0 <0.8.0;` (see for example the `OVM_CanonicalTransactionChain` contract). However, contracts that do not set explicit visibility in their constructors were only allowed starting in Solidity 0.7, meaning that it will not be possible to compile them with older versions.

Consider reviewing and updating the `pragma` statements of all contracts throughout the code base to ensure they can actually be compiled with the expected versions.

[N03] Fragile default values in Merkle tree

The `getMerkleTree` function of `Lib_MerkleTree` library fills unbalanced trees with default values. These value are chosen to simulate the effect of padding the input `_elements` with zero values to ensure the number of elements is a power of 2. Although this implicitly introduces new elements into the Merkle tree, they cannot be referenced in the corresponding `verify` function as long as their index falls outside the acceptable bound. This behavior is acknowledged, noted in the function comments, and respected throughout the code base.

Nevertheless, we understand it would be more gas-efficient and easier to reason about if the default values were replaced with a constant value that provably has no known pre-image. Consider introducing this simplification.

[N04] Gas inefficiencies

This is a non-comprehensive list of simple gas inefficiencies detected as a side-product of the audit for the development team's consideration.

In the `OVM_CanonicalTransactionChain` contract:

- The address of the Sequencer could be resolved before entering the loop to avoid unnecessary external calls during execution of the `appendQueueBatch` function. Similarly, the address referencing the queue should be resolved before entering the loop, then replacing the call to `getQueueElement` with `_getQueueElement`.

In the `OVM_StateCommitmentChain` contract:

- The `appendStateBatch` function reads from storage the chain's total elements twice (in lines 136 and 152) when it could do it once.

In the `OVM_ExecutionManager` contract:

- The `_createContract` function reads from the `messageContext.ovmADDRESS` state variable three times, when it could do it just once at the beginning of the function.

In the `OVM_L1ETHGateway` contract:

- The `_safeTransferETH` function of the `OVM_L1ETHGateway` contract uses a `new bytes(0)` object as a parameter when executing the `call` function. Using `""` in place of `new bytes(0)` achieves the same effect and saves gas. Alternatively, consider entirely replacing the custom `_safeTransferETH` function with the `sendValue` function available in OpenZeppelin Contracts.

[N05] Incomplete override

The `Abs_L1TokenGateway` abstract contract has a default amount of gas that is sent with the cross-domain message. The intention is to allow descendant contracts to change this value as needed, but the current code base does not support this.

Consider marking the `getFinalizeDepositL2Gas` function as `virtual` so it can be overridden. Additionally, consider marking the default value as `internal`, so it is removed from the public API when it is no longer in use.

[N06] Inconsistent name resolution

There are a few inconsistent name resolutions throughout the code base:

- The `onlyRelayer` modifier of the `OVM_L1CrossDomainMessenger` contract resolves the name "OVM_L2MessageRelayer", which should point to the `OVM_L1MultiMessageRelayer` contract.
-

The `OVM_L1MultiMessageRelayer` and `OVM_L1ETHGateway` contracts resolve the name `"Proxy__OVM_L1CrossDomainMessenger"`, while the `OVM_L2CrossDomainMessenger` contract resolves the name "OVM_L1CrossDomainMessenger". Presumably, these point at the same address.

- The name `"OVM_DecompressionPrecompileAddress"` should resolve to the address of the `OVM_ProxySequencerEntrypoint` contract.
- The `OVM_StateCommitmentChain` contract resolves the name "OVM_Proposer", but assigns it to a local variable called `sequencer`. This is because the sequencer and proposer are expected to be the same contract. However, in the interests of clarity, either the resolved name or the variable name should be modified for consistency (as suggested in the informational note **"[N31] Naming issues"**).

[N07] Inconsistent use of named return variables

Named return variables are used inconsistently. For example, while some functions in the `OVM_CanonicalTransactionChain` contract name their return variables, others do not. Consider removing all named return variables, explicitly declaring them as local variables, and adding the necessary return statements where appropriate. This should improve both explicitness and readability of the project.

[N08] Minimum nuisance gas per contract creation is charged twice

The `ovmCREATEEOA` and `safeCREATE` functions, used for creating accounts in the `OVM_ExecutionManager` contract, follow a similar pattern to calculate nuisance gas:

1. Call the `_initPendingAccount` function (see lines 537 and 1098), which internally calls the `_checkAccountLoad` function, which in turn charges nuisance gas the first time the referenced account is loaded. Here, the amount of nuisance gas charged includes the minimum gas (dictated by the `MIN_NUISANCE_GAS_PER_CONTRACT` constant).
2. Call the `_commitPendingAccount` function (see lines 551 and 1125), which internally calls the `_checkAccountChange` function, which in turn charges nuisance gas the first time the referenced account is changed. Here, the amount of nuisance gas charged also includes `MIN_NUISANCE_GAS_PER_CONTRACT` (as in the first case).

This means that during account creation, the minimum amount of nuisance gas tracked in the `MIN_NUISANCE_GAS_PER_CONTRACT` constant is charged twice. This appears to be unnecessary since during fraud proof verification the code for the created account would only need to be provided once. Yet it could also be argued that in the fraud proof verification the pre-state of the empty account would need to be proved as well, and that is why the `MIN_NUISANCE_GAS_PER_CONTRACT` is charged twice.

To avoid confusions, consider explicitly specifying and documenting the intended behavior, including related unit tests if appropriate.

[N09] Redundant validations during state batch deletion

The `deleteStateBatch` function of the `OVM_StateCommitmentChain` contract verifies that the passed batch header is valid using the `_isValidBatchHeader` function, and after additional checks, it executes the internal `_deleteBatch` function. Within the `_deleteBatch` function, the logic checks that both the index and batch header are valid. Yet these last two validations are redundant, since as mentioned, the `_isValidBatchHeader` function covers them both and was already executed prior to the internal call to `_deleteBatch`.

To favor simplicity and gas-efficiency, consider removing these unnecessary validations.

[N10] Contract creation can revert upon failure

Both `ovmCREATE` and `ovmCREATE2` opcodes can revert during execution (instead of returning 0 upon failure as specified in the EVM). This is due to the fact that the code is validating the deployer is allowed at the beginning of the opcodes execution (see calls to the `_checkDeployerAllowed` function here and here), instead of doing it inside the `safeCREATE` function, where similar validations are applied that do not result in a revert upon failure.

We are raising this peculiarity of the current version of the OVM as an informative note for completeness, since we understand that the development team is fully aware of this undocumented behavioral difference with EVM, and is planning to fix it by removing the deployer whitelist in the short term. Should that not be the case, consider this note of higher priority and explicitly document the described behavior in external documentation to raise user awareness.

[N11] Typographical errors

In `Lib_MerkleTree.sol`:

- In line 117, "sibline" should say "sibling".

In `Lib_MerkleTrie.sol`:

- In line 275, "31" should say "32".

In `Lib_Math.sol`:

- In line 14, "minumum" should say "minimum".

In `iOVM_ChainStorageContainer.sol`:

- In line 104, "meaing" should say "meaning".

In `OVM_CanonicalTransactionChain.sol`:

- In line 344, "minnet" should say "mainnet".
- In line 973, "que" should say "queue".
- In line 1011, "elemtent" should say "element".

In `OVM_CrossDomainEnabled.sol`:

- In line 14, "recieve" should say "receive".

In `OVM_L2ToL1MessagePasser.sol`:

- In line 9, "facilitates" is misspelled. It also includes the repeated phrase "of the".

In `Abs_L1TokenGateway.sol`:

- In line 128, "recipient's" is misspelled.

In `OVM_L1ERC20Gateway.sol`:

- In line 18, "takes" should say "take".

In `OVM_ExecutionManager.sol`:

- In line 168, "awlways" should say "always".
- In lines 191, 202, 213, and 1701, "minnet" should say "mainnet".
- In lines 1804, 1809, 1814, "unnecessary the SSTORE" should say "the unnecessary SSTORE".

In `OVM_ECDSAContract.sol`:

- In lines 142, "transfer" is misspelled.

In `OVM_FraudVerifier.sol`:

- In line 210, "minnet" should say "mainnet".

In `Lib_RLPReader.sol`:

- In lines 384 and 410, "a address" should say "an address".

[N12] Negative overflow of uint256 type

To favor readability, in line 1890 of `OVM_ExecutionManager.sol` consider replacing the negative overflow operation of an `uint256` value with the expression `type(uint256).max`.

[N13] Unnecessary return statement

Consider removing the `return` keyword from the `setGlobalMetadata` function of the `OVM_ChainStorageContainer` contract, as the `setExtraData` function of the `Lib_RingBuffer` library being called does not return any value.

[N14] Unused imports

To improve readability and avoid confusion, consider removing the following unused imports.

- In the `OVM_StateCommitmentChain` contract, the `iOVM_FraudVerifier` interface.
- In the `OVM_ExecutionManager` contract, the `OVM_ECDSAContractAccount` and `OVM_DeployerWhitelist` contracts.
- In the `OVM_L2DepositedERC20` contract, the `iOVM_L1TokenGateway` interface.
- In the `OVM_ETH` contract, the `Lib_AddressResolver` contract.
- In the `OVM_StateTransitioner` contract, the `iOVM_BondManager` interface.
- In the `Lib_OVMCodec` library, the `Lib_BytesUtils` library.

[N15] Unused events

The `iOVM_L2ToL1MessagePasser` interface defines the `L2ToL1Message` event, which is never emitted in the child contract `OVM_L2ToL1MessagePasser`.

To avoid confusion and favor simplicity, consider removing all definitions of events that are not expected to be emitted.

[N16] Unused functions

Functions `toUint24`, `toUint8` and `toAddress` of the `Lib_BytesUtils` library are never used, and can therefore be removed.

[N17] Transaction hashes might not be unique in the Canonical Transaction Chain

The `enqueue` function of the `OVM_CanonicalTransactionChain` contract constructs transaction hashes with the caller's address, the L2 target, the transaction's gas limit and its data. Since this is not enough to ensure uniqueness of hashes (that is, it could be possible to construct two transactions that result in the same hash), these transactions are instead identified by their position in the queue. However, this internal subtlety of the Canonical Transaction Chain is not explicitly documented, and might lead to

errors in off-chain services tracking transactions in the Canonical Transaction Chain, since transaction hashes are commonly assumed to be unique.

Consider including developer-friendly documentation stating how transaction hashes in the Canonical Transaction Chain are constructed, and how they should not be relied on to uniquely identify transactions.

[N18] Cross-domain messengers can be impersonated

The `relayMessage` functions of the `OVM_L1CrossDomainMessenger` and `OVM_L2CrossDomainMessenger` contracts allow relaying arbitrary cross-domain messages. Ultimately, this means that it is possible for anyone to make these contracts execute arbitrary calls (see calls here and here). Therefore, there are two scenarios developers should consider when building and integrating bridges between layer 1 and 2. Aiming for simplicity, in the following we explain both scenarios starting on layer 1 - yet a similar behavior can be seen in the opposite direction.

The simplest case would be sending a message from a user-controlled layer 1 account to any layer 2 account. In practice, this will allow the layer 1 account to make the `OVM_L2CrossDomainMessenger` contract call any target address with arbitrary data. Therefore, layer 2 accounts should be aware that they can receive arbitrary calls from the `OVM_L2CrossDomainMessenger` contract that are fully controlled by layer 1 accounts. For example, this allows any layer 1 account to steal any tokens deposited in the `OVM_L2CrossDomainMessenger` contract, or maliciously register the `OVM_L2CrossDomainMessenger` contract in the `ERC1820Registry` contract. It must be noted that during the call from the `OVM_L2CrossDomainMessenger` contract to the target address, the target address can query the `xDomainMessageSender` function to inspect the address of the layer 1 account that originated the message.

Going further, now into the second case, a subtle behavior of the `OVM_L1CrossDomainMessenger` and `OVM_L2CrossDomainMessenger` contracts allows anyone not only to *send* messages via these contracts, but also to *originate* messages from them. The execution steps to originate an L2-to-L1 message from the `OVM_L2CrossDomainMessenger` would develop as follows:

1. A user-controlled account sends a message from layer 1 calling the `sendMessage` function of the `OVM_L1CrossDomainMessenger` contract. The `target` of this message should be the `OVM_L2CrossDomainMessenger` contract, and the `message` should be an abi-encoded call to the target's `sendMessage` function, including the arbitrary data the user wants the `OVM_L2CrossDomainMessenger` contract to send to layer 1.
2. The L1-to-L2 message sent by the user is enqueued in the Canonical Transaction Chain as a regular OVM transaction.
3. In layer 2, the `relayMessage` function of the `OVM_L2CrossDomainMessenger` contract is called to relay the user's message. Following how the message was constructed, this will trigger a call from the `OVM_L2CrossDomainMessenger` contract to its own `sendMessage` function. In other words, the `OVM_L2CrossDomainMessenger` contract sends a message from L2 to L1 with target and data arbitrarily decided by the user in (1).
4.

After the fraud proof window is over, the L2-to-L1 message sent by the `OVM_L2CrossDomainMessenger` contract is relayed in layer 1.

5. The target contract in layer 1 receives a message with the user-controlled data. Should the target address query the `xDomainMessageSender` function to inspect the address of the layer 2 account that originated the message, it would receive the address of the `OVM_L2CrossDomainMessenger` contract.

The actual consequences of the described behaviors will ultimately depend on the contracts receiving these arbitrary calls, and that is why we are only reporting this as an informative note. Developers should be fully aware of these scenarios and be ready to implement the necessary defensive measures to mitigate impact on their systems. We suggest the Optimism team to include specific developer-friendly documentation highlighting this note, so as to raise awareness of the subtleties of cross-domain communication.

[N19] Subtleties of calling contracts under construction and abstracted EOAs

In the EVM, calling accounts with no executable code (that is, contracts during construction or externally owned accounts) results in an immediate halt with a STOP opcode (see subsection 9.4 of the yellow paper), and the call is considered successful. In the OVM, this behavior is not exactly replicated, due to some fundamental differences between the EVM and the OVM.

Calling abstracted EOAs

The OVM offers native account abstraction. In other words, the only type of account is smart contracts, and the closest one can get to the behavior of EOAs is implemented in the `OVM_ECDSAContractAccount` contract. As a result, "calling an EOA" is translated to calling a specific instance of this contract, and any call that does not match the selector of the `execute` function will result in an out-of-gas error.

Calling contracts under construction

In the L1 sandboxed execution environment of the OVM, calling a contract under construction results in a call to the zero address (instead of a call to the address of the contract). This is due to the fact that the internal `_callContract` function of the `OVM_ExecutionManager` contract resolves the specified target address to the actual address of the contract in L1, using the `_getAccountEthAddress` function. This results in a call to the `getAccountEthAddress` function of the `OVM_StateManager` contract, which will return the zero address, because the contract under construction has not been yet committed to the state. More specifically, during a creation operation an account is first initialized as pending without setting its L1 address, then created, and finally committed. Therefore any address resolution before the account is committed will resolve to the zero address. The call to the zero address will be successful, and execution will simply continue. Off-chain services tracing the internal execution of fraud proof verifications might find this behavior relevant, as they will see a call to the zero address where they would have expected a call to a contract under construction.

It should be noted that both scenarios described were raised as issues of Medium severity in our November 2020 report as **"[M03] Call to contract in construction results in call to the zero address"** and **"[M04] Calls to abstracted EOA accounts may result in Out of Gas error"**. Taking into account that we did not receive specific feedback on those issues, and that the behavior of the system remains, we assume that the Optimism team has acknowledged these scenarios and consider them intended. Therefore, we are only describing them in this informative note for completeness, and to suggest explicitly documenting them either with inline comments, docstrings, external developer documentation or system specification if the Optimism team considers it appropriate.

[N20] Repeated authentication logic in State Manager

The `isAuthenticated` function of the `OVM_StateManager` contract can be used to validate whether a given address is allowed to write into the contract's state. The same functionality is implemented in the `authenticated` modifier.

To avoid code repetition, consider modifying the `authenticated` modifier so that it calls the `isAuthenticated` function to determine if the caller is authenticated. This note can be disregarded should the current implementation be more favorable in terms of gas costs.

[N21] Not using available bytes32 utilities

To favor simplicity and favor reusability, consider replacing the operations to cast from and to `bytes32` types in lines 76, 86, 100 and 110 of `OVM_ProxySequencerEntrypoiny.sol` with the available utilities in the `Lib_Bytes32Utils` library.

[N22] Missing operations in Execution Manager wrapper library

The `Lib_SafeExecutionManagerWrapper` library offers functions to facilitate writing OVM safe code that can be compiled using the standard Solidity compiler. However, it is missing a number of wrappers for OVM operations, namely:

- `ovmCREATE2`
- `ovmSTATICCALL`
- `ovmEXTCODEHASH`
- `ovmEXTCODECOPY`
- `ovmL1TXORIGIN`
- `ovmL1QUEUEORIGIN`
- `ovmGASLIMIT`
-

- `ovmNUMBER`
- `ovmTIMESTAMP`

Consider including wrapper functions for them in the `Lib_SafeExecutionManagerWrapper` library.

[N23] Unnecessary standalone contract to relay multiple messages

The `OVM_L1MultiMessageRelayer` contract implements a single function `batchRelayMessages` that forwards multiple cross-domain messages to the `OVM_L1CrossDomainMessenger` contract. To favor simplicity, this function could be moved to the `OVM_L1CrossDomainMessenger` contract (where it should be marked with the `onlyRelayer` modifier).

[N24] Data returned by relayed message is ignored

The `relayMessage` function of the `OVM_L2CrossDomainMessenger` contract executes an `ovmCALL` to the message's target. While the call's success flag is validated to determine whether the message can be considered successful, the returned data is never stored nor logged. As a consequence, relevant returned data from relayed messages will not be accessible after a message is relayed. The described behavior also occurs in the `relayMessage` function of the `OVM_L1CrossDomainMessenger` contract.

Should this be the intended behavior of the `relayMessage` functions, consider documenting it in the functions' docstrings. Otherwise, consider including a mechanism to retrieve the returned data of relayed messages, which could include storing the data in storage, or logging it using an event.

[N25] Duplicated code in cross-domain messenger contracts

The `relayMessage` functions of the `OVM_L1CrossDomainMessenger` and `OVM_L2CrossDomainMessenger` contracts behave in a similar way and share a non-trivial amount of duplicated logic.

To reduce code duplication and favor reusability, consider abstracting away repeated logic to an internal function of the `Abs_BaseCrossDomainMessenger` parent contract.

[N26] Duplicated code for memory copy utility

The `_copy` function of the `Lib_RLPReader` library and `_memcpy` function of the `Lib_RLPWriter` serve a similar purpose: copying pieces of memory to other locations. To reduce duplicated logic and ease maintenance, consider defining a single memory copy utility function, and reusing it throughout the code base. Alternatively, to favor a simpler implementation, consider using the identity precompile contract at address `0x0000000000000000000000000000000000000004` to copy pieces of memory.

[N27] Redundant check when proving contract state

Within the `proveContractState` function of the `OVM_StateTransitioner` contract, there is a `require` statement that checks two conditions. The first condition, `hasAccount`, checks whether the account's code hash is non-zero. The second condition, `hasEmptyAccount`, checks that the account's code hash matches the `EMPTY_ACCOUNT_CODE_HASH` hash.

Since the `EMPTY_ACCOUNT_CODE_HASH` is non-zero, the second condition implies the first one. Therefore, consider removing the call to `hasAccount`.

[N28] Base contract not marked as abstract

Contracts that are not intended to be instantiated directly, such as `OVM_CrossDomainEnabled`, should be marked as abstract to favor readability and avoid unintended usage.

[N29] Inconsistent coding style

Some instances of inconsistent coding style were identified in the code base. Specifically:

- While most `internal` and `private` functions explicitly denote their visibility by prepending their names with an underscore, functions `getCrossDomainMessenger` and `sendCrossDomainMessage` of the `OVM_CrossDomainEnabled` contract fail to do so.

To favor readability, consider always following a consistent style throughout the code base. We suggest using Solidity's Style Guide as a reference.

[N30] Lack of explicit visibility in state variables

The following state variables and constants are implicitly using the default visibility.

In the `OVM_ECDSAContractAccount` contract:

- The `EXECUTION_VALIDATION_GAS_OVERHEAD` and `ETH_ERC20_ADDRESS` constants.

In the `OVM_ProxyEOA` contract:

- The `IMPLEMENTATION_KEY` constant.

In the `Abs_L2DepositedToken` contract:

- The `DEFAULT_FINALIZE_WITHDRAWAL_L1_GAS` constant

To favor readability, consider explicitly declaring the visibility of all state variables and constants.

[N31] Naming issues

To favor explicitness and readability, several parts of the contracts may benefit from better naming. Our suggestions are:

- The `FRAUD_PROOF_WINDOW` state variable should be renamed to `FRAUD_PROOF_WINDOW_IN_SECONDS`.
- The `SEQUENCER_PUBLISH_WINDOW` state variable should be renamed to `SEQUENCER_PUBLISH_WINDOW_IN_SECONDS`.
- In the `_appendBatch` function, a local variable `sequencer` is assigned the address resolved for the `OVM_Proposer` key. As the proposer might not be the sequencer (their roles having been split in PR#252), the local variable name should be modified to avoid confusion.

[N32] Implicit casting operations

- The `sendMessage` function of the `Abs_BaseCrossDomainMessenger` contract implicitly upcasts its `_gasLimit` parameter from `uint32` to `uint256` when it's passed to the call to `_sendXDomainMessage`.
- The `execute` function of the `OVM_ECDSAContractAccount` contract implicitly casts a `uint64` number to `uint256` when validating that the transaction's nonce matches the expected one.

For added readability, consider making casting operations explicit where possible.

Conclusions

4 critical and 4 high severity issues were found. Several changes and recommendations were proposed to reduce the code's attack surface and improve its overall quality.