certora

Security Assessment & Formal Verification Report

AAVE

Custom CCIP-1.5.1 TokenPool contracts for GHO

Dec-2024

Table of content

Project Summary

Project Scope

Project Name	Repository (link)	Latest Commit Hash	Platform
Modified CCIP-1.5.1	aave-ccip	49caffa	EVM/Solidity 0.8

Project Overview

This document describes the specification and verification of the **Modified CCIP-1.5.1** using the Certora Prover and manual code review findings. The work was undertaken from **23 Dec 2024** to **02 Jan 2025**.

The scope of our review is the modifications that were done in the following contracts:

- UpgradeableBurnMintTokenPool
- UpgradeableBurnMintTokenPoolAbstract
- UpgradeableLockReleaseTokenPool
- UpgradeableTokenPool

The Certora Prover demonstrated that the implementation of the Solidity contracts above is correct with respect to the formal rules written by the Certora team. In addition, the team performed a manual audit of all the Solidity contracts. During the verification process and the manual audit, no bug was discovered.

Protocol Overview

The contracts under review are customized versions of the Chainlink Cross-Chain Interoperability Protocol (CCIP) contracts, specifically designed for the GHO cross-chain strategy. These include the UpgradeableBurnMintTokenPool and UpgradeableLockReleaseTokenPool, which are based on the CCIP's BurnMintTokenPool and LockReleaseTokenPool from the 1.5.1. version, respectively. The modifications made are as follows:

1. Soften Solidity compiler version of all base contracts of BurnMintTokenPool and LockReleaseTokenPool to ensure compatibility with the gho-core contracts.
2. Addition of upgradeability mechanism using the battle-tested transparent proxy pattern.
3. Introduction of a bridge limit on UpgradeableLockReleaseTokenPool to regulate the maximum amount of tokens that can be transferred out (either burned or locked) from Ethereum to other

chains, thereby enhancing UX by enabling the precise calculation of GHO liquidity available for minting on remote networks.

4. Addition of authorized functions to directly mint/burn liquidity, thereby increasing/reducing the facilitator's bucket level.

Coverage

1. With respect to formal verification we ran the existing tests for the contract UpgradeableLockReleaseTokenPool, and added several additional rules. See more information later.
2. With respect to manual auditing we went over the ARFC and the code and we saw that the implementation matches the described changes. We mainly focused on the bridged limit which sets a limit to the amount of token that can be bridged from Ethereum to other chains.
3. UpgradeableTokenPool contract:
   Description: The UpgradeableTokenPool contract serves as a foundational component within the CCIP architecture. Primarily inheriting from the original contract, it incorporates minimal modifications to preserve existing functionalities while introducing mechanisms to support future enhancements.
   Reserved Storage Space (__gap):
   - **Purpose:** A __gap array has been added to reserve storage slots, accommodating potential future variable additions without necessitating shifts in the storage layout down the inheritance chain.
   - **Rationale:** This proactive measure ensures that upgrades can be performed seamlessly, preventing storage collisions and maintaining contract integrity.
   - **Impact:** While the inclusion of __gap introduces slight overhead, it is a standard practice in upgradeable contracts to safeguard against storage-related issues during upgrades. Given that Chainlink Labs Limited (CLL) does not anticipate changes to the storage layout with CCIP version 1.6, the addition of __gap is both safe and purposeful.

   Configuration Handling:
   The constructor(...) initializes all necessary variables, eliminating the need for an explicit decimals check. This omission is justified as CCIP is GHO-specific, operating with 18 decimals across all chains.
   Router Initialization:
   The router is initialized within the inherited contracts (UpgradeableLockReleaseTokenPool and UpgradeableBurnMintTokenPool), obviating the need for router configuration within the UpgradeableTokenPool itself.
4. UpgradeableLockReleaseTokenPool contract:
   Description: The UpgradeableLockReleaseTokenPool contract extends the UpgradeableTokenPool to incorporate mechanisms for locking and releasing tokens across chains. It includes functionalities to manage liquidity and ensure the secure migration of tokens during upgrades.

releaseOrMint(...):

- **Security Checks:** Before releasing or minting tokens, the contract performs essential validations to ensure the legitimacy and correctness of the transaction.
- **Bridged Amount Management:** The function ensures that the amount to be released does not exceed the currently bridged amount (s_currentBridged). If valid, it decrements s_currentBridged accordingly.
- **State Management:** Reducing s_currentBridged before performing validations ensures internal state consistency. Since state changes occur before external interactions, the approach maintains security and integrity.
- **Data Trustworthiness:** The reliance on trusted source pool data eliminates the risk of invalid decimal information affecting the local amount calculation.

setCurrentBridgedAmount(...):

- This function was introduced as a separate setter rather than updating the s_currentBridged value through each liquidity operation, preserving Chainlink's original implementation. It is meant to be used only during migrations by the executor (the contract owner) to adjust on-chain records accurately. This design ensures minimal modification to the pool's core logic while retaining a controlled, owner-only mechanism for special migration scenarios.

provideLiquidity(...):

- **Access Control:** Only the designated rebalancer can add liquidity, ensuring that only authorized entities manage the pool's funds.
- **Liquidity Addition:** Upon validation, the function transfers the specified amount of tokens from the sender to the pool, emitting an event for transparency.
- **Bridge Amount Management:** The function does not directly alter s_currentBridged. Instead, adjustments are handled via a separate setter. provideLiquidity(..) will be used only on contract upgrade thus the need for the use of the setter can be contained.

withdrawLiquidity(...):

- **Access Control:** Similar to providing liquidity, only the rebalancer can withdraw funds, maintaining strict oversight over liquidity movements.
- **Liquidity Withdrawal:** The function verifies sufficient liquidity before transferring tokens back to the sender, emitting an event to log the removal.
- **Rug Pull Mitigation:** By restricting withdrawals to the rebalancer—a trusted address—the risk of unauthorized liquidity removal is minimized, safeguarding the pool's funds.

5. UpgradeableBurnMintTokenPool contract:
   Description: The UpgradeableBurnMintTokenPool contract specializes in minting and burning tokens.
   directBurn(...):

- **Access Control:** Restricted to the contract owner, ensuring that only authorized entities can perform burns.
- **Functionality:** Burns a specified amount of tokens, aiding in the migration process by reducing the facilitator's bucket level.
- **Safety of Burn Function:** The burn function securely reduces the facilitator's bucket level and is appropriately restricted to the owner (executer), mitigating risks associated with unauthorized burns.

directMint(...)

- **Access Control:** Restricted to the contract owner, ensuring that only authorized entities can perform mints.
- **Functionality:** Mints a specified amount of tokens directly to a target address, typically the old token pool or a facilitator being offboarded. This action, when coupled with the corresponding burn in the old pool, seamlessly transfers the facilitator's bucket level without creating unbacked tokens.
- **Safety of the Mint Function:** By limiting the mint function to owner-only access and coordinating it with an equivalent burn, the total token supply remains intact, preventing the creation of unbacked tokens and mitigating risks associated with unauthorized mints.

6. Ownable2Step contract:
   Description: The Ownable2Step contract manages ownership transitions within the upgradeable contracts. It facilitates a secure and verifiable process for transferring ownership, ensuring that only authorized entities can assume control.
   _transferOwnership(...):
   - **Validation:** Ensures that ownership is not inadvertently transferred to the caller, preventing accidental loss of control.
   - **State Update:** Sets the pending owner and emits an event to log the ownership transfer request.
   - **Access Control Refinement:** By changing the visibility of the _transferOwnership function from private to internal, the contract maintains encapsulation while allowing derived contracts to manage ownership transitions securely.
7. AIP Proposal to Replace Existing GHO Token Pools in CCIP:
   Existing GHO pools are deprecated in favor of a new, more secure version (v1.5.1).

   Key reasons include stronger security, better configurability (e.g., rate and bridge limits), and reliable ownership transfer during migrations.
   - **Atomic Minting & Burning:** The old pool's entire bucket level is minted in the new pool and then immediately burned in the old pool.

- **Integrity Preservation:** This approach ensures there are never unbacked tokens, maintaining supply accuracy throughout migration.
- **Direct Liquidity Transfer:** Liquidity moves from the old pool to the new one, ensuring the new pool can handle redemptions right away.
- **Rebalancer Role & Bridge Limits:** The new pool is set as the rebalancer to finalize liquidity migration, while the old pool's bridge limit is set to zero to prevent further bridging operations.
- **Accepting Ownership:** The new pools transfer ownership to the executor.
- **Parameter Configuration:** Key token pool parameters, such as rateLimitAdmin and bridgeLimitAdmin, are set during the setup phase.

Formal Verification

Verification Notations

Formally Verified	The rule is verified for every state of the contract(s), under the assumptions of the scope/requirements in the rule.
Formally Verified After Fix	The rule was violated due to an issue in the code and was successfully verified after fixing the issue
Violated	A counter-example exists that violates one of the assertions of the rule.

Formal Verification Properties

In the table below we specify all the formally verified rules that we wrote for the contract UpgradeableLockReleaseTokenPool, and give a detailed description for them. A link to the Certora's prover report can be found here.

P-01. currentBridge_LEQ_bridgeLimit

Status: Verified

Rule Name	Status	Description	Rule Assumptions
currentBridge_LEQ_bridgeLimit	Verified	The value of the variable s_currentBridged is less or equal to the value of the variable s_bridgeLimit.	This rule can be violated if one calls the function setBridgeLimit(...) with a value smaller than s_currentBridged.

P-02. withdrawLiquidity_correctness

Status: Verified

Rule Name	Status	Description
withdrawLiquidity_correctness	Verified	The rule checks that while calling the function withdrawLiquidity(), the balance of the contract changes accordingly.

P-03. provideLiquidity_correctness

Status: Verified

Rule Name	Status	Description
provideLiquidity_correctness	Verified	The rule checks that while calling the function provideLiquidity(), the balance of the contract changes accordingly.

P-04. only_bridgeLimitAdmin_or_owner_can_call_setBridgeLimit

Status: Verified

Rule Name	Status	Description
only_bridgeLimitAdmin_or_owner_can_call_setBridgeLimits	Verified	The rule checks that only the bridgeLimitAdmin or the owner can call the function setBridgeLimits().

P-05. only_lockOrBurn_can_increase_currentBridged	
Status: Verified	

Rule Name	Status	Description
only_lockOrBurn_can_increase_currentBridged	Verified	The rule checks that the only function that can increase the value of s_currentBridged is lockOrBurn().

P-06. only_releaseOrMint_can_decrease_currentBridged	
Status: Verified	

Rule Name	Status	Description
only_releaseOrMint_can_decrease_currentBridged	Verified	The rule checks that the only function that can decrease the value of s_currentBridged is releaseOrMint().

P-07. only_owner_can_call_setCurrentBridgedAmount	
Status: Verified	

Rule Name	Status	Description

P-08. only_rebalancer_can_call_provideLiquidity

Status: Verified

Rule Name	Status	Description
only_rebalancer_can_call_provideLiquidity	Verified	The rule checks that only the rebalancer can call the function provideLiquidity().

P-09. only_rebalancer_can_call_withdrawLiquidity

Status: Verified

Rule Name	Status	Description
only_rebalancer_can_call_withdrawLiquidity	Verified	The rule checks that only the rebalancer can call the function withdrawLiquidity().

Disclaimer

The Certora Prover takes a contract and a specification as input and formally proves that the contract satisfies the specification in all scenarios. Notably, the guarantees of the Certora Prover are scoped to the provided specification and the Certora Prover does not check any cases not covered by the specification.

Even though we hope this information is helpful, we provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the contract is secure in all dimensions. In no event shall Certora or any of its employees be liable for any claim, damages, or other liability, whether in an action of contract, tort, or otherwise, arising from, out of, or in connection with the results reported here.

About Certora

Certora is a Web3 security company that provides industry-leading formal verification tools and smart contract audits. Certora's flagship security product, Certora Prover, is a unique SaaS product that automatically locates even the most rare & hard-to-find bugs on your smart contracts or mathematically proves their absence. The Certora Prover plugs into your standard deployment pipeline. It is helpful for smart contract developers and security researchers during auditing and bug bounties.

Certora also provides services such as auditing, formal verification projects, and incident response.