certora

Security Assessment & Formal Verification Report

Sonic Gateway

November 2024

Prepared for SonicLabs

Table of content

Project Summary......3
Project Scope......4
Project Overview......4
Findings Summary......5
Severity Matrix......5
Detailed Findings......6
Medium Severity Issues......7
M-01 potential DoS of payFastLane() feature......7
M-02 deposit() and withdraw() DoS......8
M-03 High validators ID can lead to timeouts or API exhaustion......9
M-04 High number of validators replacements could lead to out-of-gas error......11
M-05 Unregistering token can lock user's funds until the bridge is dead......13
Low Severity Issues......14
L-01 Lack of input validation when adding or removing validators......14
L-02 Token registering mistake can't be reversed......15
L-03 Possible token pair mismatch between L1 and L2......15
L-04 Weak validation for fastLanePadding variable......16
Informational Issues......18
I-01. Public key length check......18
I-02. Confusing naming......18
I-03. mintedToOriginal variable is never used......18
I-04. Typos......19
Formal Verification......20
Verification Notations......20
General Assumptions and Simplifications......20
Formal Verification Properties......21
Bridge......21
P-01. Bridge solvency......21
P-02. It's impossible to claim on L2 the same deposit twice......21
P-03. It's impossible to withdraw from L2 the same deposit twice......22
P-04. One (except REGISTER_ROLE) cannot front-run a withdrawal from L2 of another user......22
P-05. No one can front-run and block the claiming of funds......22
P-06. After claiming funds on L2, it's always possible to withdraw them right after......23
P-07. Only the users have control of their funds......23
Token Deposit......24
P-01. TokenDeposit solvency......24
P-02. Correct behavior of a dead bridge......25
P-03. It's impossible to claim on L1 the same withdrawal twice......26
P-04. It's impossible to deposit the same deposit ID twice......26

P-05. No one can front-run and block the claiming of funds.. 27
P-06. No one can front-run and block a deposit...27
Token Pairs.. 28
P-01. Token Pairs are unique and invertible...28
Disclaimer.. 29
About Certora.. 29

Project Summary

Project Scope

Project Name	Repository (link)	Latest Commit Hash	Platform
Fantom Foundation Sonic Bridge	Sonic Bridge - Github	558465d3aba2ffae1a4436a2fc14c723b82926df Fix commit: 1b36d4d3f2165e94e6a5c7480e9ff6b0f03fc795	EVM

Project Overview

This document describes the specification and verification of **Sonic Bridge** using the Certora Prover and manual code review findings. The work was undertaken from **26/9/2024** to **14/10/2024**

The following contract list is included in our scope:

```
contracts/
cmd/
internal/
```

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, the Certora team discovered bugs in the Solidity contracts code, as listed on the following page.

Findings Summary

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

Severity	Discovered	Confirmed	Fixed
Critical	-	-	-
High	-	-	-
Medium	5	5	4
Low	4	4	0
Informational	4	4	3
Total	**13**	**13**	**7**

Severity Matrix

Impact	High	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Low	Low	Medium
		Low	Medium	High
		Likelihood		

Detailed Findings

ID	Title	Severity	Status
M-01	Potential DoS of payFastLane() feature	Medium	Fixed
M-02	deposit() and withdraw() DoS	Medium	Fixed
M-03	High validators ID can lead to timeouts or API exhaustion	Medium	Not fixed
M-04	High number of validators replacements could lead to out-of-gas error	Medium	Fixed
M-05	Unregistering token can lock user's funds until the bridge is dead.	Medium	Fixed
L-01	Lack of input validation when adding or removing validators	Low	Not fixed
L-02	Token registering mistake can't be reversed	Low	Not fixed
L-03	Possible token pair mismatch between L1 and L2	Low	Not fixed
L-04	Weak validation for fastLanePadding variable	Low	Not fixed

Medium Severity Issues

M-01 potential DoS of payFastLane() feature		
Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: UpdateManager.sol	Status: Fixed	

Description:

UpdateManager.sol smart contracts is required to have 0 balance for *payFastLane()* function to be called successfully.

```
JavaScript
function payFastLane(uint256 blockNum) external payable {
       require(address(this).balance == msg.value, "Fast lane busy"); // balance must be 0
before this tx
       ...
}
```

On the other hand, an admin *update()* function clears any sent ether to the smart contract. However a griefer can send 1 wei every time admins call *update()* through *selfdestruct*.

Exploit Scenario: A griefer sends 1 wei everytime *UpdateManager's* balance is cleared.

Recommendations: Since non-zero balance acts like mutex that guards the fast lane feature, we suggest implementing such a mechanism by introducing a new variable.

Customer's response: Acknowledged and fixed.

Fix Review: Fixed at https://github.com/Fantom-foundation/Bridge/commit/0bbedf5aaef07d4ea2349fdd485f55745dc7f0a1 .

M-02 deposit() and withdraw() DoS		
Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: TokenDeposit.sol Bridge.sol	Status: Fixed	Violated Property: P-06. No one can front-run and block deposit function.

Description:

Users can choose their deposits id arbitrarily.

This allows for them to be frontrun by a malicious actor using the same id making their original transaction revert.

The same applies for the withdraw() function in Bridge.sol.

```JavaScript
function deposit(uint256 id, address token, uint256 amount) external {
...
...
function withdraw(uint256 id, address token, uint256 amount) external {
```

Recommendations: We recommend assigning the user an id that is not totally arbitrary, and therefore, cannot be front run.

Fix Review: Fixed at https://github.com/Fantom-foundation/Bridge/commit/e26f9bfd4d32fb540bedc1f14b9f15aab5a01450#diff-d641d27f43a4e1d3dc731dff26fbbaf62e42388ae1455ec70b9521a1e6f27868R64 .

M-03 High validators ID can lead to timeouts or API exhaustion

Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: ValidatorsRegistry.sol	Status: Not Fixed	

Description:

validatorsCount() tracks the highest ID **ever** registered (never decreases).

If the ID becomes too high (for example 10000 - even if there are only 10 validators), it can lead to web3 provider timeouts or API exhaustion or any other issue related to web3 provider rate limits.

In file *validators.go* a for loop iterates over *i=0..maxCount*. In each spin of the loop there are 4 Web3 provider read calls (2 validator address reads and 2 validator weight reads). With max id = 10000 there will be 40000 read calls to the web3 provider which likely would either exhaust the API credits or introduce temporary back-off limiting the API calls.

```
C/C++
       sourceCountBig, err := source.ValidatorsCount(&sourceOpt)
       [...]
       destCountBig, err := dest.ValidatorsCount(&destOpt)
       [...]
       sourceCount := sourceCountBig.Int64()
       destCount := destCountBig.Int64()

       maxCount := sourceCount
       [...]

       for i := int64(1); i <= maxCount; i++ { // validators indexed from 1
              [...]
```

```
                sourceAddr, err = source.ValidatorAddress(&sourceOpt, id) // web3 API call
                [...]
                destAddr, err = dest.ValidatorAddress(&destOpt, id) // web3 API call
                [...]
                sourceWeight, err := source.ValidatorWeight(&sourceOpt, sourceAddr) // web3 API
call
                [...]
                destWeight, err := dest.ValidatorWeight(&destOpt, destAddr) // web3 API call
                [...]
        }
```

To make matters worse, there is nothing preventing *setValidators()* with an ID much larger than the current length. For example, setting ID to *type(uint256).max*. Therefore unless the input is sanitized so ID can't be a very large arbitrary value, there could be very big holes in the array, leading to timeouts or API exhaustion even if in reality the number of validators is a lot lower.

Current implementation can be characterized as **eager check** because it loops over the entire possible range to collect changes. Alternative would be to have **lazy update** i.e. listening for specific on-chain events (in this case, validator addition or deletion) and reacting on this specific event.

Exploit Scenario:

Validator id is set too high. Even if this is by mistake it can't be reversed.

validators.go function *LookupValidatorsChanges()* is DoSed.

Because this function is present in most hot paths, the majority of the off-chain system will be down

Recommendations: We recommend implementing a validator container that doesn't contain holes or/and implementing lazy update.

Customer's response: Acknowledged, won't fix.
In the production deployment, validators are synced from the SFC contract (through SFCUpdateVerifier), which assigns IDs sequentially.

M-04 High number of validators replacements could lead to out-of-gas error		
Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: ValidatorsRegistry.sol	Status: Fixed	

Description:

If there has been a big change in validators, the function `LookupValidatorsChanges()` could return an array that would lead to an out of gas revert, when its content reaches the `for` loop in SFCUpdateVerifier.verifyUpdate() line 65:

`for (uint16 i; i < signatures.length; i++) `.

```JavaScript
if sourceAddr != destAddr || sourceWeight.Cmp(destWeight) != 0 {
                    updated = append(updated, contracts.IUpdateVerifierValidator{
                            Id:     id,
                            Addr:   sourceAddr,
                            Weight: sourceWeight,
                    })
            }
```

```JavaScript
        for (uint16 i; i < signatures.length; i++) {
            address signer = ECDSA.recover(messageHash, signatures[i]);
            uint256 signerWeight = validatorWeight(signer);
            require(signerWeight > 0, "Invalid signer");
            require(signer > lastSigner, "Invalid signatures order");
            lastSigner = signer;
            weight += signerWeight;
            signers[i] = validatorId(signer);
        }
```

This is respectively how much gas this function would use according to different numbers of validators:
10 validators: gas used 1005494.

100 validators: gas used 8584776.

1000 validators: gas used 86201101.

10000 validators: gas used 914256397.

Source: https://gist.github.com/hake-certora/44606e34cdc3a66793914943c6307c2a

Recommendations: We recommend either:

1. Parsing the validators in batches when calling *verifyUpdate()*, rather than calling it with all the validators changes at once.
2. Setting a limit to the `IUpdateVerifier.Validator[] memory newValidators` array in ValidatorRegistry.sol line 25, so the absolute number of validator changes never gets too big to generate an out of gas error.

Customer's response: Acknowledged and fixed.

Fix Review: Fixed at https://github.com/Fantom-foundation/Bridge/commit/9d817b66e0857a7f738f56821c1350a25bef34cd#diff-4f390d94a22d3c12c3babe6b16adb730b539d15a92ca39d16129d9a9e25267bbR27 .

M-05 Unregistering token can lock user's funds until the bridge is dead.		
Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: TokenPairs.sol Bridge.sol	Status: Fixed	Violated Property: P-05. No one can front-run and block the claiming of funds.

Description:

If a user deposited tokens in L1 and before Withdrawing them in L2 the token is unregistered (tokenPairs.unregister) then the funds are stuck in the bridge. The user can't even cancel the deposit on L1 because the bridge is not dead and this is the only scenario when a user can cancel his deposit.

Bridge.sol:

```JavaScript
        address mintedToken = ITokenPairs(tokenPairs).originalToMinted(token);
        require(mintedToken != address(0), "No minted token for given token");
```

Exploit Scenario: Admin unregisters token and the claim function always reverts and the outcome is the user's funds are stuck in the bridge.

Recommendations: Either don't unregister if there are funds to be claimed or the better way is to add functionality to cancel deposits for disabled tokens.

Customer's response: The `unregister` function is removed in this commit and thus solves the issue.

Low Severity Issues

L-01 Lack of input validation when adding or removing validators		
Severity: **Low**	Impact: **Medium**	Likelihood: **Low**
Files: UpdateManager.sol	Status: Not Fixed	

Description:

When deleting validators it's better to check if newWeights is exactly 0, otherwise the totalWeight will be off which can lead to *UpdateManager.update()* always reverting.

Recommendations: Consider requiring weight equal zero when removing validator

Customer's response: Acknowledged. There is no deleting in the ValidatorsRegistry - only setting the weight to 0. (Since we are going to be consistent with SFC.) When the weight is set, the original weight of the validator is subtracted from the "total" before adding the newWeight.

Fix Review: Won't be fixed.

L-02 Token registering mistake can't be reversed		
Severity: **Low**	Impact: **High**	Likelihood: **Low**
Files: TokenPairs.sol	Status: Not Fixed	

Description:

If a token is mistakenly paired, it can never be reset or changed.

Exploit Scenario: Admin accidentally set the wrong token address. The contract needs to be redeployed.

Recommendations: It is recommended to allow token overwriting.

Customer's response: Acknowledged. Thought about introducing unregistering tokens functionality, but decided to discard it.

Fix Review: Won't be fixed.

L-03 Possible token pair mismatch between L1 and L2		
Severity: **Low**	Impact: **High**	Likelihood: **Low**
Files: TokenPairs.sol	Status: Not Fixed	

Description:

Input error could lead to a mismatch between original and minted tokens between L1 and L2.

Exploit Scenario: User would deposit WETH, and get to claim WETH2 on L2. However, on Bridge.sol WETH does not have a corresponding pair, line 53 reverts: `require(mintedToken != address(0), "No minted token for given token");` and the user is not able to claim. Leading to frozen funds.

Recommendations: We recommend to verify the storage slots on L1 using the merkleProof to ensure there is no discrepancy between L1 and L2.

Customer's response: Acknowledged. Not fixed.

L-04 Weak validation for *fastLanePadding* variable

Severity: **Low**	Impact: **High**	Likelihood: **Low**
Files: UpdateManager.sol	Status: Not Fixed	

Description:

In *setFastLanePadding()* current requirement is that fastLanePadding needs to be smaller than heartbeat. However, it needs to be at least 50% smaller than heartbeat. Otherwise fast lane feature is DoSed

Exploit Scenario:

If admin sets fastLanePadding that is too big then either of these 2 checks would revert

```javascript
require(blockNum > lastStateUpdate + fastLanePadding, "Block number too low");
require(blockNum < nextStateUpdate - fastLanePadding, "Block number too high");
```

Recommendations: There should be a stronger requirement: fastLanePadding needs to be at least 50% smaller than heartbeat. Otherwise payFastLane is DoSed.

Customer's response: Acknowledged. FastLanePadding can be set only by the administrator. It is valid for an administrator to disable the fast lane entirely.

Fix Review: Won't fix.

Informational Issues

I-01. Public key length check

Description: In *validatorAddress(uint256)* in contract *SFCUpdateVerifier.sol*, one should check the exact length of the public key to be 66 instead of zero (line 30), to make sure the copy loop doesn't revert relatively late.

Customer's response: Acknowledged and fixed.

Fix Review: Fix at https://github.com/Fantom-foundation/Bridge/commit/30081434cfe900dc144ff3e9a3095a1e2c7373d8 .

I-02. Confusing naming

Description:

1. *validatorsCount()* returns max ID **ever** registered. It doesn't return the number of validators. And it doesn't even return the current max ID.
2. *getDeadStatus()* is not a getter. It actually fetches the most recent value and stores it. Consider renaming to fetchDeadStatus()

Recommendation: Consider renaming above functions

Customer's response: Since we never delete a validator (only set his weight to zero), it does return the max validator ID.
Fix Review: Fixed at https://github.com/Fantom-foundation/Bridge/commit/a23a07de7558c3e1a796672867d3a98382ddbef3 .

I-03. mintedToOriginal variable is never used

Description: The variable `mintedToOriginal` in TokenPairs.sol is never used. Therefore, it should be removed to save gas and improve code readability.

Customer's response: It is used for consistency checks - to avoid possibility a single minted token is mapped to multiple L1 tokens.

Fix Review: Won't fix.

I-04. Typos

Description: Both TokenDeposit.sol (line 59) and Bridge.sol (line 39) have a typo in the documentation referring to the `claim()` function where the word Claim is missing the letter "i": /// Clam -> /// Claim

Customer's response: Acknowledged and fixed.
Fix Review: Fixed at https://github.com/Fantom-foundation/Bridge/commit/339e4b22ceb47fad9fb981a33c201355d10e8358 .

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.

General Assumptions and Simplifications

1. We used Solidity Compiler version 8.27 to verify the protocol.
2. In the course of verifying the Bridge and TokenDeposit functionality, We have summarized the necessary function calls for verifying the bridge message utilizing a Merkle tree structure. While We have assumed that the Merkle tree proof implementation is correct, without directly verifying its correctness, We have ensured that other critical components of the Bridge and TokenDeposit have been properly implemented and validated. These implementations, combined with the assumption of a secure Merkle tree proof, are sufficient for the overall functionality and security of the bridge to operate as expected.

3. In addition to summarizing the message verification process using a Merkle tree, we have also assumed that the validation process carried out by the validators is correct and functions as intended. Specifically, we rely on the assumption that the validators properly execute their role in ensuring the integrity and correctness of the Bridge and TokenDeposit, including accurately validating transactions and the associated Merkle proofs.

4. We assumed the Bridge works only with OpenZeppelin ERC20s and MintedERC20s.

Formal Verification Properties

Bridge

Module General Assumptions

- We assume that all loops iterate at most three times.

Module Properties

P-01. Bridge solvency	
Status: Verified	

Rule Name	Status	Description	Link to rule report
preservationOfMintedToken	Verified	*The conservation of tokens in the Bridge component.*	Report

P-02. It's impossible to claim on L2 the same deposit twice.	
Status: Verified	

Rule Name	Status	Description	Link to rule report
cantClaimTwice	Verified	*This rule verifies that it is impossible to claim on L2 the same deposit twice.*	Report

P-03. It's impossible to withdraw from L2 the same deposit twice.	
Status: Verified	

Rule Name	Status	Description	Link to rule report
cantWithdrawTwice	Verified	*This rule verifies that it is impossible to withdraw on L2 the same deposit twice.*	[Report]()

P-04. One (except REGISTER_ROLE) cannot front-run a withdrawal from L2 of another user.	
Status: Formally Verified After Fix	Assumptions: only trusted parties can unregister tokens

Rule Name	Status	Description	Link to rule report
withdrawFr	Formally Verified After Fix	*This rule verifies that the withdraw method can be front-run only by using the same ID.* *[M-02 deposit() and withdraw() DoS]()*	[Report]()

P-05. No one can front-run and block the claiming of funds.	
Status: Violated	Assumptions:

Rule Name	Status	Description	Link to rule report
claimFr	Violated	*This rule verifies that the claim method can't be front-run.* *[M-05 Unregistering token can lock user's funds until the bridge is dead.]()*	[Report]()

P-06. After claiming funds on L2, it's always possible to withdraw them right after.	
Status: Verified	Assumptions:

Rule Name	Status	Description	Link to rule report
claimingSuccessfulyThenWithdrawingNeverReverts	Verified	*After claiming funds on L2, it's always possible to withdraw them right after.*	[Report]

P-07. Only the users have control of their funds.	
Status: Verified	Assumptions:

Rule Name	Status	Description	Link to rule report
claimingTokenADoesntAffectOthers	Verified	*Claiming token A assuming it is paired with token B - shouldn't affect token C (A != C && B =! C)*	[Report]
integrityOfClaim	Verified	*After a successful call for claim the right withdrawal is claimed and the user's balance is increased by the correct amount.*	[Report]
integrityOfWithdraw	Verified	*After a successful call for withdraw() the right withdrawal is set and the user's balance is decreased by the correct amount.*	[Report]
withdrawalsIdAreUserInjective	Verified	*This rule verifies that withdrawal IDs are user-injective.*	[Report]

onlyMsgSenderChangedsWithdrawStatus	Verified	*This rule verifies that Only the msg.sender can change the withdrawal ID status.*	[Report]()

Token Deposit

Module General Assumptions

- We assume that all loops iterate at most three times.

Module Properties

P-01. TokenDeposit solvency	
Status: Verified	Assumption: we ignored the withdrawWhileDead() function as it is a privileged operation so we consider it to be trusted, meaning this operation can violate the invariant in theory but we trust it would be done responsibly.

Rule Name	Status	Description	Link to rule report
ContractBalanceAtLeastSumOfUnclaimedAndDeposited	Verified	*The contract ERC20 balance is bigger or equal to the sum of all unclaimed plus the sum of deposited.*	[Report]()

P-02. Correct behavior of a dead bridge.	
Status: Verified	

Rule Name	Status	Description	Link to rule report
onceDeadAlwaysDead	Verified	*Once the bridge dies, it remains dead.*	Report
noDepositsWhileDead	Verified	*No deposits while bridge is dead => deposit[id]Before != deposit[id]After => deposit[id]After == 0*	Report
integrityOfDeposit	Verified	*A successful call for deposit means the bridge was live and sets the deposit in the correct ID and updates the erc20 balances correctly.*	Report
integrityOfWithdrawWhileDead	Verified	*A successful call for withdrawWhileDead means the bridge was dead and updates the erc20 balances correctly.*	Report
integrityOfCancelDepositWhileDead	Verified	*Can cancel deposits only when the bridge is dead.*	Report
balanceNotIncreasedWhileBridgeIsDead	Verified	*contract erc20 balance shouldn't increase while the bridge is dead.*	Report
cannotCallMethodsWhenBridgeIsAlive	Verified	*Verifies that one cannot call "dead" methods while the bridge is alive.*	Report
canOnlyDepositWhenAlive	Verified	*Verifies that one can call deposit() only when the bridge is alive.*	Report

P-03. It's impossible to claim on L1 the same withdrawal twice.

Status: Verified

Rule Name	Status	Description	Link to rule report
cantClaimTwice	Verified	*It's impossible to claim on L1 the same withdrawal twice.*	Report
onlyMsgSenderChangedsDepositStatus	Verified	*Only the msg.sender can change the deposit id status.*	Report

P-04. It's impossible to deposit the same deposit ID twice.

Status: Verified

Rule Name	Status	Description	Link to rule report
cantDepositTwice	Verified	*It's impossible to deposit the same deposit ID twice.*	Report
depoistsIdAreUserInjective	Verified	*Deposits IDs are user-injective.*	Report
NonEmptyDepositIsAttributedToUser	Verified	*A non-zero deposit hash is attributed to the correct user.*	Report

P-05. No one can front-run and block the claiming of funds.	
Status: Verified	Assumptions: The contract has enough tokens to pay the users (solvent).

Rule Name	Status	Description	Link to rule report
claimFr	Verified	*No one can front-run and block the claiming of funds.*	[Report]()

P-06. No one can front-run and block a deposit.	
Status: Formally Verified After Fix	Assumptions: The bridge is not dead (if the bridge is updated as dead just before we re-run this is an expected behavior)

Rule Name	Status	Description	Link to rule report
depositFr	Formally Verified After Fix	*This rule verifies no one can front-run and block deposit function.* *[M-02 deposit() and withdraw() DoS]()*	[Report]()
NonEmptyDepositIsAttributedToUser	Verified	*A non-zero deposit hash is attributed to the correct user.*	[Report]()

Token Pairs

Module General Assumptions

- We assume that all loops iterate at most three times.

Module Properties

P-01. Token Pairs are unique and invertible.	
Status: Verified	

Rule Name	Status	Description	Link to rule report
originalToMinted Uniqueness	Verified	*This rule verifies that if token A != token B => originalToMinted(A) != originalToMinted(B)*	Report
mintedToOriginal Uniqueness	Verified	*This rule verifies that if token A != token B => mintedToOriginal(A) != mintedToOriginal(B)*	Report
mintedToOriginal Invertability	Verified	*This rule verifies that mintedToOriginal is invertible*	Report
originalToMinted Invertability	Verified	*This rule verifies that originalToMinted is invertible.*	Report
originalToMinted PairingValidity	Verified	*If an original token is not paired to a minted token, then no minted token is paired to this original token.*	Report
mintedToOriginal PairingValidity	Verified	*If a minted token is not paired to an original token, then no original token is paired to this minted token.*	Report
mintedOfZeroIsZ ero	Verified	*No paired original token to address zero.*	Report
originalOfZeroIs Zero	Verified	*No paired minted token to address zero.*	Report

Disclaimer

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.