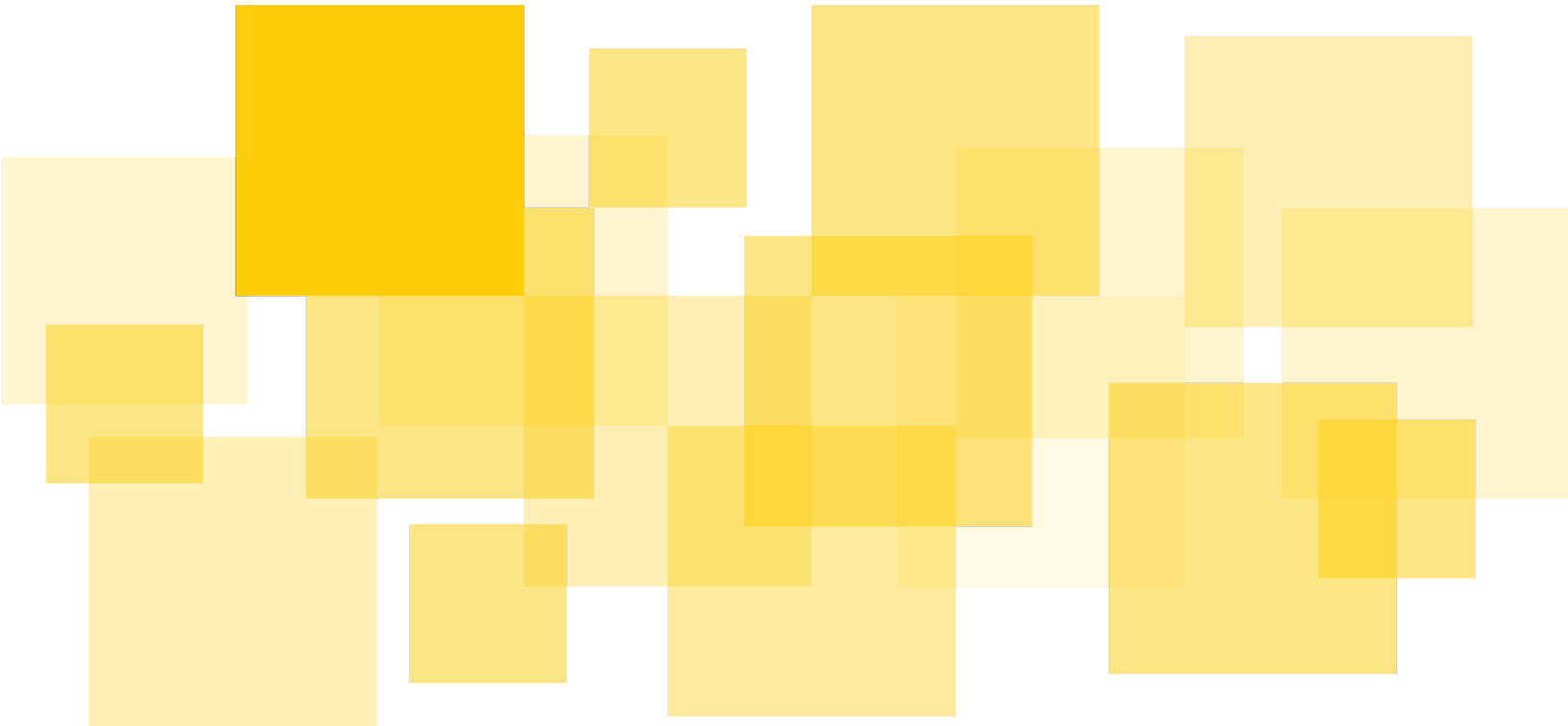


Formal Verification Report

Wormhole

Delivered: 2023-05-16



Prepared for Wormhole Foundation by Runtime Verification, Inc.





[Disclaimer](#)

[Introduction](#)

[Foundry + KEVM Workflow](#)

[Specification](#)

[Fuzzing](#)

[Symbolic Execution](#)

[Methodology](#)

[Properties](#)

[Getters](#)

[Governance](#)

[GovernanceStructs](#)

[Implementation](#)

[Messages](#)

[Setters](#)

[Setup](#)

[Shutdown](#)



Disclaimer

This report does not constitute legal or investment advice. The preparers of this report present it as an informational exercise documenting the due diligence involved in the secure development of the target contract only, and make no material claims or guarantees concerning the contract's operation post-deployment. The preparers of this report assume no liability for any and all potential consequences of the deployment or use of this contract.

Smart contracts are still a nascent software arena, and their deployment and public offering carries substantial risk. This report makes no claims that its analysis is fully comprehensive, and recommends always seeking multiple opinions and audits.

This report is also not comprehensive in scope, excluding a number of components critical to the correct operation of this system.

The possibility of human error in the manual review process is very real, and we recommend seeking multiple independent opinions on any claims which impact a large quantity of funds.



Introduction

The Wormhole Foundation has engaged Runtime Verification for a project on using mechanized formal verification on the Wormhole code base using KEVM. The goal is to create Foundry fuzz tests for desirable properties of some core contracts, then use KEVM's Foundry integration to prove that these tests will pass under all circumstances. Normally, formal verification with KEVM requires building out a set of assumptions and theorems that allow the underlying K prover to automatically prove the desired specifications (or disprove them and find a bug). With the Foundry integration, the majority of the K definitions required by the prover can be automatically generated from the Foundry tests.

The targets for this engagement have been the Ethereum core messaging contracts, located at <https://github.com/wormhole-foundation/wormhole/tree/main/ethereum/contracts> (excluding subdirectories). The version of the code under the scope of this engagement was frozen at commit [f9758b38e7efd9d7e5b21a48e009c65e2138710c](https://github.com/wormhole-foundation/wormhole/commit/f9758b38e7efd9d7e5b21a48e009c65e2138710c).

Foundry + KEVM Workflow

The KEVM-Foundry integration re-uses the property tests written for Foundry as formal specifications for KEVM proofs. The main advantage of this approach is that specifications can be written and read in Solidity and no further knowledge of a dedicated specification language is required to follow along. Another benefit of this method is that we can run the property tests before we try to prove them symbolically. This gives us a faster feedback cycle. The flowchart below visualizes the high-level workflow, which can be split into three stages.

Specification


The initial step of the workflow consists in writing property tests that capture the properties that we wish to verify. These tests assert that, for every test input that satisfies some assumptions, the results follow some expected behavior (revert with some error message, emit an event, return a value that satisfies some conditions, etc.). These tests serve as our formal specification.

Fuzzing

The first feedback cycle then consists in running the tests as fuzz tests using Foundry, by randomly-generating concrete inputs and executing the test over them. This allows us to quickly identify issues on the code or on the test itself, and either fix the code or refine the test (for example by adding missing assumptions). In this way, this step is important not only to find an initial set of bugs, but also to iterate on the formal specification and ensure that it accurately reflects the desired property.

Symbolic Execution

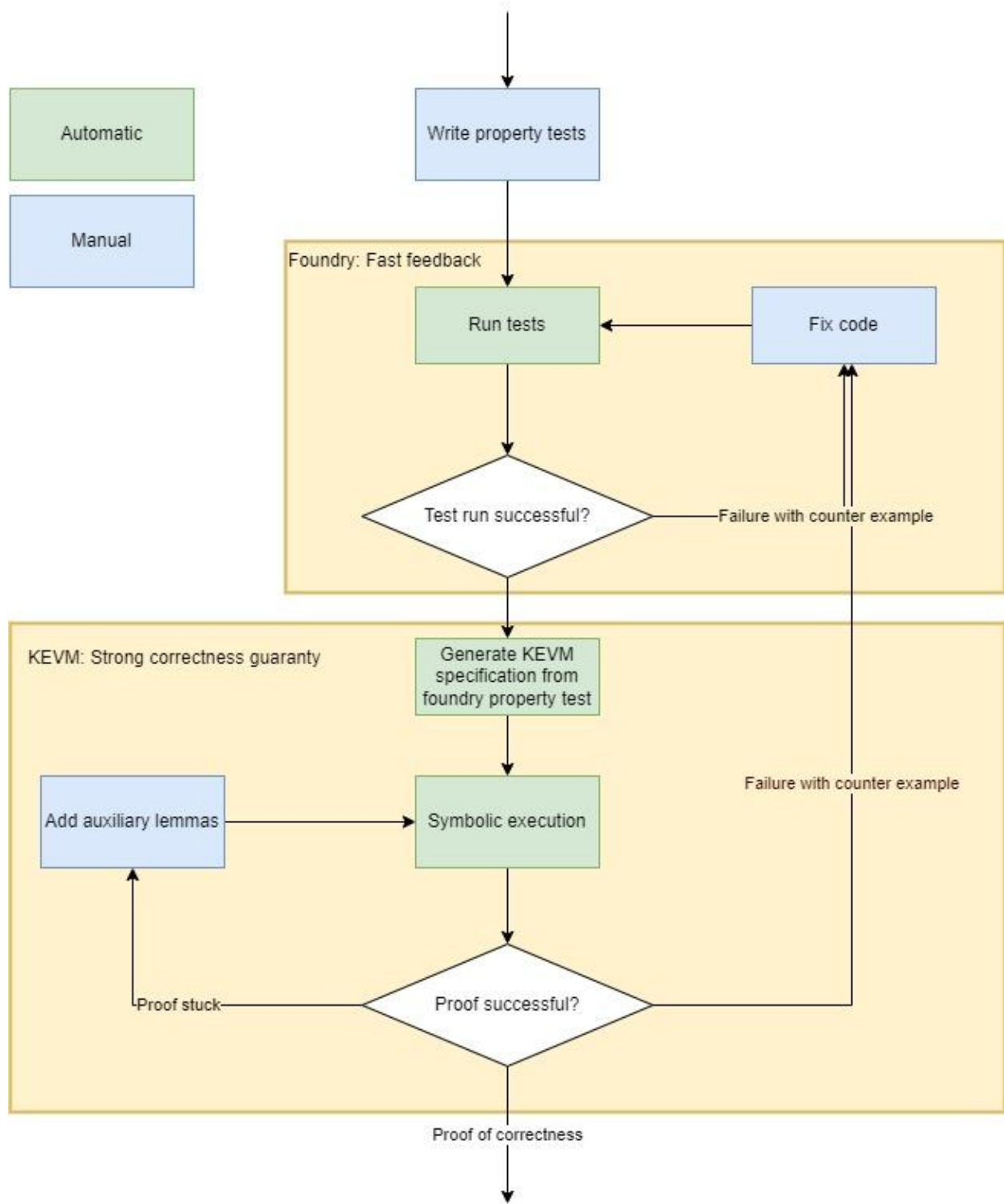
After all tests are passing, the process can move on to symbolic execution using KEVM. In contrast to fuzzing, symbolic execution uses symbolic rather than concrete inputs. This means that the input values are interpreted as mathematical variables that can contain any value that satisfies the assumptions. The symbolic execution engine then executes the code over these symbolic variables, building a mathematical expression capturing the set of possible values that they can take at each point in the execution. In this way, KEVM is able to explore the entire input space of the test without having to run it concretely for every input. At the end of the execution, the engine checks if every possible final state passes the test. If so, then we can state that the test passes for every possible input, and therefore the code has been formally verified to satisfy the specification. If there is a final state where the test fails, then this might indicate a bug in the code, in which case it needs to be fixed and the process repeated.



In some cases, however, KEVM might not be able to complete verification, but also not find an actual bug in the code. There are a few different scenarios that can happen:

- The engine reaches a final state where the test fails, but this state cannot be reached in practice. This happens because KEVM was not able to prove that this execution path is impossible.
- The engine gets stuck in a non-final state of the execution where it is unable to make progress. KEVM works by matching the current symbolic expression to a set of internal rules, and if it cannot find a rule that matches the current expression it cannot continue execution. This usually means that it was not able to simplify the expression to the form it needs.
- The symbolic execution branches too much, creating too many execution paths that the engine cannot complete in a reasonable time. Like before, this usually means that KEVM was not able to prove that some of these paths are impossible. In some cases, however, the branches are valid execution paths. If there are too many paths that need to be considered, it might be necessary to make simplifying assumptions to narrow the scope of the verification and focus on only a single, main path or small set of paths.

When the symbolic execution gets stuck or is unable to prune a spurious execution path, KEVM requires manual intervention in order to make progress. This is done by adding lemmas, auxiliary rules that can be fed into the tool to tell it how an expression can be simplified. KEVM ships with a body of lemmas, but new lemmas might need to be added for a specific proof. Any lemmas that we have written in the process of the verification are provided along with the tests.



Properties

The sections below summarize the properties for which we have written Foundry tests, split by contract. Some of the tests were able to be formally verified by symbolically executing them using KEVM¹, while others were only executed as fuzz tests using Foundry. The following table lists which is the case for each of the test contracts written during the engagement:

TestGetters	KEVM
TestGovernance	FUZZING
TestGovernanceStructs	KEVM
TestImplementation	FUZZING
TestMessagesRV	FUZZING
TestSetters	KEVM
TestSetup	KEVM
TestShutdown	FUZZING

For the Getters, Setters and GovernanceStructs tests, we have two versions of the test: the base version meant to be executed as a fuzz test in Foundry and a version with the `_KEVM` suffix meant to be symbolically executed using KEVM. This version is a wrapper for the original test which additionally uses two KEVM-specific cheatcodes:

- `symbolicStorage` is used to assume that the relevant contract is in an arbitrary state prior to executing the test, in order to ensure that verification is fully general and doesn't depend on specific storage values.
- `infiniteGas` is used to assume an unlimited amount of gas for executing the test, allowing the symbolic execution to avoid out-of-gas checks while also computing a symbolic expression for gas consumption.

¹ KEVM version corresponding to commit [f5c1795aea0c7d6781c94f0e6d4c434ad3ad1982](https://github.com/ethereum/kevm-abi/commit/f5c1795aea0c7d6781c94f0e6d4c434ad3ad1982).

As these cheatcodes aren't recognized by Foundry, the KEVM versions of the test should not be executed using `forge test`. They can be excluded by passing the option `--no-match-test .*_KEVM`.

In the case of the tests for the Setup contract, which specifically test properties relating to how Wormhole is initialized, we instead used a concrete set-up simulating this initialization procedure, with fixed values of the `chainId`, `evmChainId`, `module`, `governanceChainId` and `governanceContract` fields. As a simplifying assumption, we used a fixed guardian set with a single guardian. A similar concrete set-up with a single guardian was used for the remaining test contracts that were only executed as fuzz tests, as in this case we cannot use the `symbolicStorage` KEVM cheatcode to fully generalize the contract state. The only exception is the `TestMessagesRV` test contract, which generalizes the pre-existing tests in `TestMessages` to an arbitrary number of guardians (up to 19).

Getters

The Getters contract includes the following functions:

- `getGuardianSet`
- `getCurrentGuardianSetIndex`
- `getGuardianSetExpiry`
- `governanceActionIsConsumed`
- `isInitialized`
- `chainId`
- `evmChainId`
- `isFork`
- `governanceChainId`
- `governanceContract`
- `messageFee`
- `nextSequence`

Except for `getGuardianSet` (which has a dynamic return type of `Structs.GuardianSet` memory, and therefore would require a loop invariant for verification) and

`getGuardianSetExpiry` (which is unused in practice²), we have written Foundry tests and verified the following properties for the getters above:

- The getter returns the current content of the corresponding bytes of storage.
- All storage slots remain unchanged.

Governance

The Governance contract includes the following functions for submitting governance actions:

- `submitContractUpgrade`
- `submitSetMessageFee`
- `submitNewGuardianSet`
- `submitTransferFees`
- `submitRecoverChainId`

For each of these functions, we have written Foundry tests for the following properties and run them on fuzzed inputs³:

- If the function receives a valid VAA, it performs the appropriate governance action and the storage is updated accordingly.
- The function does not update any other storage slots.
- The function reverts if the VAA is for a different module, chain or fork (`submitRecoverChainId` can only be called on a fork, while the others cannot be called on a fork).
- The function reverts if the VAA does not originate from the governance chain, was not emitted by the governance contract, or was not signed by the current guardian set.
- The function reverts if it is called twice for the same VAA.

² The field is stuck at 0 in the deployed Wormhole contract (as can be seen on [Etherscan](#)), and the `expireGuardianSet` setter instead uses the constant 86400 for the expiration interval. To maintain the storage layout, this variable cannot be removed, but there are a few possible ways to address this inconsistency:

1. Upgrade the contract and set `guardianSetExpiry` to 86400. Preferably, the `expireGuardianSet` function should also be updated to use this variable instead of the hard-coded constant. This also opens the possibility of adding a setter function so governance can change the expiry time in the future.
2. Otherwise, document that this field is unused and consider changing its name to reflect this. Also consider either removing the getter or having it return the constant 86400 instead.

³ The fuzzed inputs are used to build a valid VAA. However, some of the fields to build a valid VAA need to be concrete, so they can match the values used in the `setup` function

We have also tested the following additional properties for specific functions:

- `submitContractUpgrade` emits the `ContractUpgraded` event if called successfully.
- `submitContractUpgrade` initializes the new implementation, and therefore if `initialize` is called again it reverts with the “already initialized” error message.
- `submitNewGuardianSet` reverts if the new guardian set is empty or if the new guardian set index is not incrementing by 1.

GovernanceStructs

The `GovernanceStructs` contract includes the following functions:

- `parseContractUpgrade`
- `parseGuardianSetUpgrade`
- `parseSetMessageFee`
- `parseTransferFees`
- `parseRecoverChainId`

Excluding `parseGuardianSetUpgrade`, which includes a loop and therefore would require a loop invariant for verification, we have written Foundry tests and verified the following properties for the functions above:

- A sequence of bytes in the correct format is parsed correctly by the function into the corresponding governance action. By “correct format” we mean:
 - It is the correct length (99 bytes for `parseTransferFees`, 67 bytes for the others).
 - It has the correct value in the bytes corresponding to the action field.
- If the function is given a sequence of bytes that has the right length but whose action field corresponds to a different action, it reverts with the appropriate error message.

The following additional property would require a loop invariant to be verified for sequence of bytes of arbitrary length. Therefore, we have only executed the test on concrete inputs via fuzzing, not symbolically using KEVM:

- The function reverts if it is given a sequence of bytes of the wrong length.

Implementation

We have written and run Foundry tests for the following properties of the `publishMessage` function of the `Implementation` contract:

- If called successfully, the function increases the sequence number of the caller by 1.
- The function does not modify any other storage slots.
- If called successfully, the function emits the `LogMessagePublished` event.
- The function reverts if the amount transferred is different from the message fee.

Messages

We have written generalized versions of the following pre-existing Foundry tests for the `verifySignatures` function:

- `testCannotVerifySignaturesWithOutOfBoundsSignature`
- `testCannotVerifySignaturesWithInvalidSignature` (split into 2)
- `testVerifySignatures`

Unlike the original tests, these tests initialize a guardian set with a random number of guardians up to 19, rather than a single guardian. The tests check the following scenarios:

- `testCannotVerifySignaturesWithOutOfBoundsSignature`: Start with a valid set of signatures of the same message, one by each guardian. Then, change the guardian index of one of the signatures to a number above the number of guardians. The `verifySignatures` function will revert with the "guardian index out of bounds" message.
- `testCannotVerifySignaturesWithInvalidSignature1`: Start with a valid set of signatures of the same message, one by each guardian. Then, replace one of the signatures with a fake signature formed of 65 random bytes. The `verifySignatures` function will either revert with the "ecrecover failed with signature" message or return false with the reason "VM signature invalid". Note that theoretically it is possible that the random bytes will correspond to the correct signature, but this is extremely unlikely.
- `testCannotVerifySignaturesWithInvalidSignature2`: Start with a valid set of signatures of the same message, one by each guardian. Then, replace one of the signatures with a signature of the same message by an arbitrary other address. The `verifySignatures` function will return false with the reason "VM signature invalid".

- `testVerifySignatures`: If `verifySignatures` is called with a valid set of signatures of the same message, one by each guardian, it will return `true` with the empty string as the reason.

Setters

The Setters contract includes the following functions:

- `updateGuardianSetIndex`
- `expireGuardianSet`
- `storeGuardianSet`
- `setInitialized`
- `setGovernanceActionConsumed`
- `setChainId`
- `setGovernanceChainId`
- `setGovernanceContract`
- `setMessageFee`
- `setNextSequence`
- `setEvmChainId`

Except for `storeGuardianSet`, which includes a loop and therefore would require a loop invariant for verification, we have written Foundry tests and verified the following properties for the setters above:

- The setter updates the corresponding bytes of storage to the correct value.
- All other bytes in the same storage slot remain unchanged.
- All other storage slots remain unchanged.

Additionally, for `setEvmChainId`, we have verified the following property:

- The setter reverts if the new chain ID is different from `block.chainId`.

Setup

We have written and run Foundry tests checking that calling `Setup.setUp` upgrades the implementation away from the `Setup` contract. This is done via the following two properties:

- Calling `setUp` a second time reverts with an “unsupported” message, as the underlying implementation is no longer the `Setup` contract.

- Calling `initialize` after `setUp` reverts with an “already initialized” message, as `setUp` already initializes the new implementation.

Shutdown

We have written and run Foundry tests checking the following properties of the Shutdown contract:

- After upgrading the implementation to the Shutdown contract, calling `initialize` does not revert, but also does not change the storage.
- After upgrading the implementation to the Shutdown contract, calling `publishMessage` always reverts.