Formal Verification Report

Optimism Labs

Delivered: 2022-12-23

Prepared for Optimism Labs by Runtime Verification, Inc.

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.

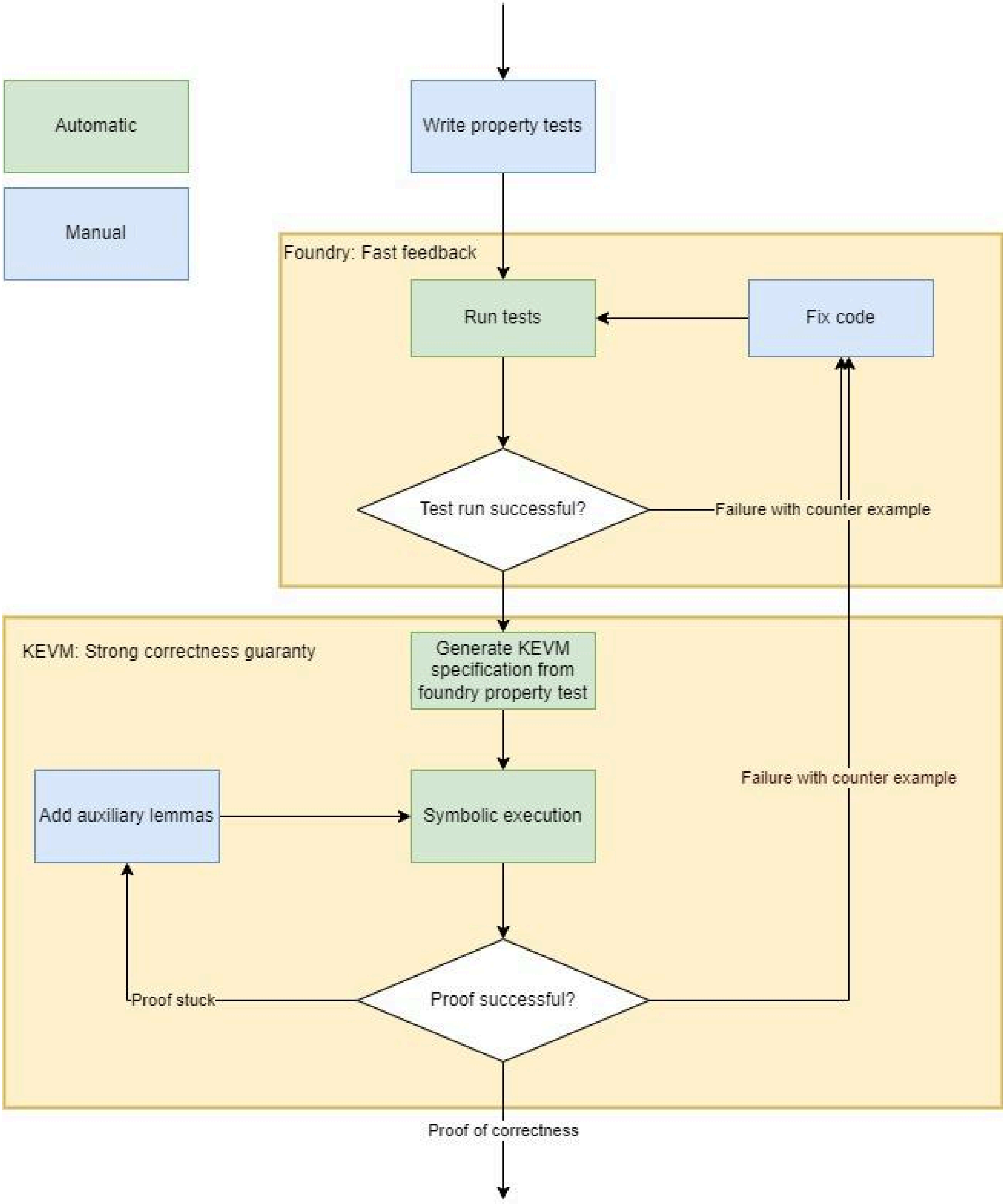
Introduction

Optimism Labs has engaged Runtime Verification for a pilot project on using mechanized formal verification on the Optimism code base using KEVM. The goal is to create Foundry fuzz tests for the full functionality of some core functions, 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). KEVM-Foundry is beta software which requires the engagement to both fine-tune the tooling as well as performing the normal proving tasks. Therefore, the engagement has been focused on both providing assumptions and theorems useful for the Optimism codebase, as well as improving the KEVM-Foundry integration.

The target for this engagement has been the `depositTransaction` function in the OptimismPortal contract. The function is crucial for censorship resistance of the Optimism rollup – it provides a mechanism for sending transactions to Optimism via transactions on Ethereum that can not be ignored by sequencers. The function does this by emitting an event. The aim is to prove or disprove that the `depositTransaction` will always emit the correct event, except under a finite and enumerable set of conditions of obviously incorrect invocations.

Foundry+KEVM Workflow

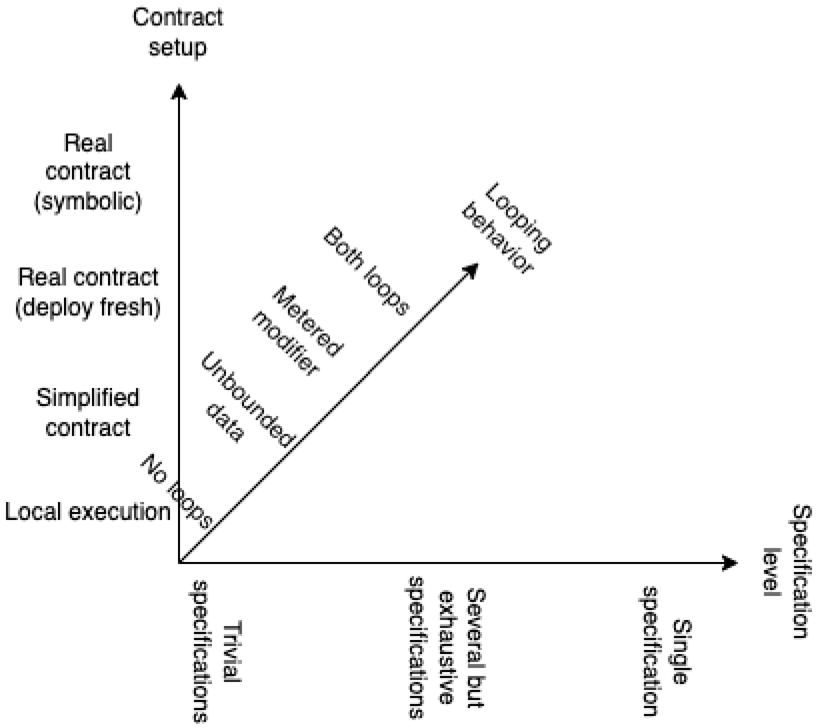
The KEVM-Foundry integration re-uses the property tests written for Foundry as formal specifications for KEVM proofs. The main advantage of this approach is that specifications can be written and read in Solidity and no further knowledge of a dedicated specification language is required to follow along. 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 following flowchart visualizes the high-level workflow:


Automatic
Manual
Write property tests
Foundry: Fast feedback
Run tests
Fix code
Test run successful?
Failure with counter example
KEVM: Strong correctness guaranty
Generate KEVM specification from foundry property test
Add auxiliary lemmas
Symbolic execution
Failure with counter example
Proof stuck
Proof successful?
Proof of correctness

Methodology

We have built out the proof by writing simplified versions of the Solidity function and gradually making more and more realistic versions of the function, and proving the specification of each in turn. There have been three axes to do so:

1. Contract setup – executing the code in simple or more realistic conditions.
2. Looping behavior – avoiding loops that complicate proofs.
3. Specification level – giving a full spec of the code written as a single function, or breaking it into more parts, or less complete subspecifications.



Contract setup

On the one hand, we have simplified invocations by first putting a version of the `depositTransaction` directly in the test contract, then producing a more realistic version that deploys a simplified `OptimismPortal` contract and tests against it, before testing against the full contract.

Looping behavior

One of the most complex aspects of mechanized formal verification is proving the correct termination of loops. The `depositTransaction` function seems loop-free on the surface, but the contract body contains three loops on the bytecode level. The first is the usage of `abi.encodePacked`, where the last parameter is a dynamic array of bytes, passed in as an argument to `depositTransaction` and stored in memory. The copying requires iterating over the array. Similarly, the Solidity compiler will generate a bytecode level loop when the encoded bytearray is passed as argument to the event. The third loop is in the metered modifier, which uses up remaining gas in a loop.

To simplify the proofs, we have first proved that the function behaves correctly when the metered modifier is removed and the dynamic byte array passed to depositTransaction is empty. This guarantees a loop-free execution. From there, we have set practically infinite dynamic array sizes (on the order of exabytes or more) and attempted the proofs over those conditions. Finally, we have reinstated the

Specification level

For many proofs we need to start by proving less general versions of our theorems. For example, only proving that `abi.encodePacked` works correctly, or that certain arithmetic expressions can be simplified in specific ways. These proofs can then be recombined into larger proofs of more useful theorems.

The Process

The first step was writing specifications. We started at the middle level of complexity: a mostly complete spec that is split up into a few different sub-proofs. We wrote one specification for the fact that the proofs should fail when a transaction is passed the `isCreation` flag while not being addressed to the `0x0` address, and one spec each for the cases where: it is a correct contract creation transaction from an EOA; when it is a contract creation from a contract; when it is not a creation transaction and the sender is an EOA; and one for when it is not a creation transaction and the sender is a contract. These specifications are over a newly deployed contract – which is not expected to cause any issues, because the `depositTransaction` function does not rely on any contract storage. (TODO: is this correct? check metered modifier.) There are some simplifying assumptions in these specs that will need to be generalized for the proof to be as complete and simple as possible:

1. The contract should be written to some address as already deployed, rather than being freshly deployed as part of the test.
2. The spec should be written as a single function, which covers both the revert case and all the success cases.

Conclusions

We found some relevant deviations from the specification, and we learned some limitations of the current proving environment that we have addressed and scheduled to address in future internal work.

Overall we have gained high confidence in the correctness of the depositTransaction() function, excluding the metered modifier, based on verification alone. It behaves correctly in all instances we have verified. The remaining risks are: whether the metered modifier can cause the function to exit successfully, but without emitting the event; if we have missed any edge case in our specifications; whether the deployment process of the contract introduces some bugs (making the actual executed bytecode different from the code we execute locally). The first two of these risks we consider minimal. Firstly, the metered modifier, having its bulk of logic being executed after the body of the depositTransaction(), could not likely revert before the event is emitted. The remaining possibilities for violating the specification are unspecified failures (of which we have found two) but which are of far less importance than that events are emitted on success. Secondly, it's simple to hand-verify that our specifications cover all input parameters. The third is more complex but causes us little worry – the function is not reliant on contract storage except in looking at resource usage, which is not enough to prevent a correct event from being emitted.

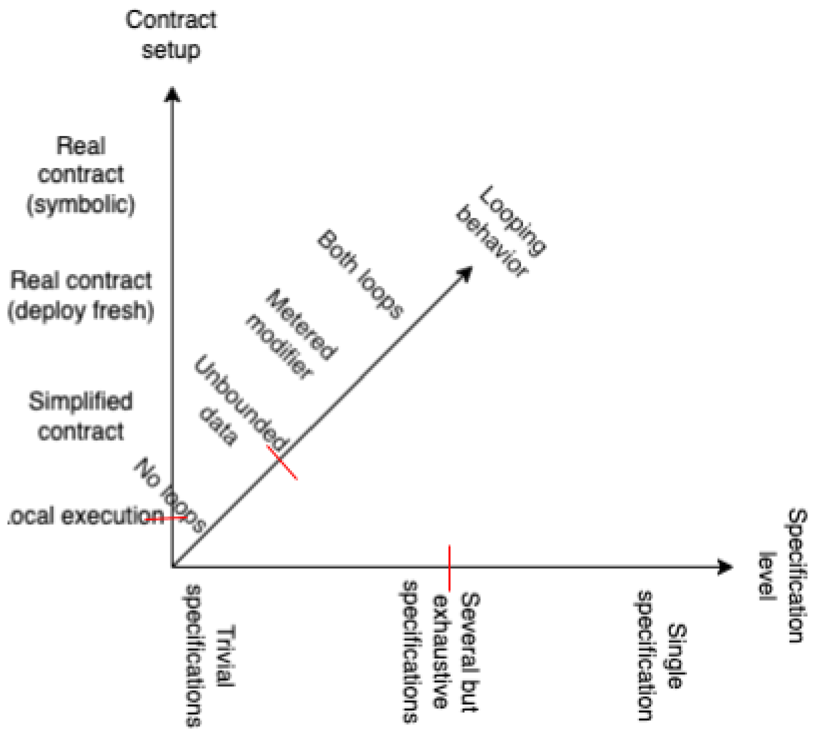
Of course, we would ideally be able to go all the way on our axes of specification and proving that specification: covering the metered modifier, giving the specification in a single function and operating on the deployed contract code with symbolic state. To reach these goals we would need to make some further refinements to KEVM-Foundry – which we have identified and started pursuing during the engagement.

Detailed findings

Successful proofs

We have proved the code correctness under the following conditions:

- Contract setup: Local execution
- Looping behavior: Unbounded data
- Specification: Several but exhaustive specifications



Our verification uses arbitrary but fixed addresses, and thus would not locate issues that could arise from calling the code from other specific addresses. However a simple thorough look at the code reveals that any address will do, and the only semantic difference is whether or not the from field should be aliased or not.

Errors in the code

We found two cases where the function deviates from the specification. Both relate to the metered modifier, and we do consider the deviations as very minor bugs.

The specification stated that the call should always emit the TransactionDeposited event, unless the calldata specifies it to be a creation event and the recipient address is not the 0x0 address. Apart from the trivial failure of this specification where not enough gas was passed and execution halts prematurely, at least the following two scenarios violate that specification.

1. If the _gasLimit parameter is set too high so that the maximum available resource limit would be exceeded, execution will fail with the error message “ResourceMetering: cannot buy more gas than available gas limit”.
2. If there has already been resources used in the current block and the caller passes too high of a _gasLimit, then the addition of params.prevBoughtGas + _amount in the metered modifier will cause an overflow and fail.

Both of these are edge cases related to bad input data and cause a failure and a revert, and thus are not dangerous to the Portal.

Gas optimizations

Proving code involving loops tends to require using what is called “loop invariants”. This makes it so that even implicit loops in the EVM end up being discovered as part of the proving process. We found an implicit loop in the use of abi.encodePacked() over an array stored in memory, but not in one stored in calldata. The loop copying data from memory is 2-3x as gas expensive as simply copying the calldata, and is harder to produce a proof over because it compiles to a loop in the EVM.

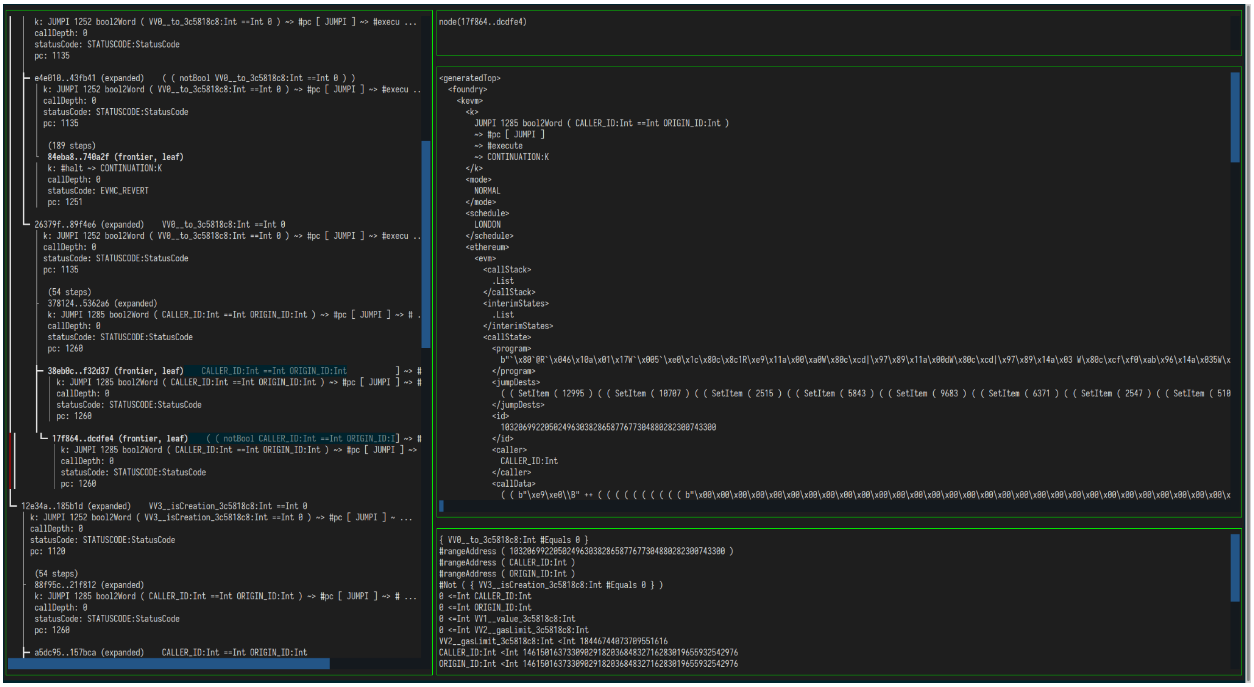
Storing the data argument in calldata instead of memory is therefore a good way to gas golf the depositTransaction function. It would also be possible to avoid abi.encodePacked() and use inline assembly to copy the relevant parameters directly from calldata, since they are unmodified within the function.

Success of proving process

The Foundry integration into KEVM is still in alpha and has improved significantly during this engagement. On the one hand, we have improved the K frameworks ability to reason over typical EVM situations, by adding more simplifications to KEVM and weeding out reasoning bugs in K.

One thing that has proven especially hard to work with is code which semantically depends on the remaining gas. When proving code correctness, it is usually best to assume infinite gas for the transaction. This means gas deductions have no real effect on the gas left (infinity - constant value = infinity). This in turn means that the symbolic expression of gas left becomes structurally simpler, which speeds up proving and improves legibility. It is also generally sound, because running out of gas causes a trivial revert that is rarely of any interest. However when the code uses the `gasleft()` function in Solidity (the GAS opcode in EVM) then infinite gas is no longer a sound or even meaningful assumption. There are a few different approaches that can be taken to get past this issue, and we are currently debating which to incorporate into KEVM.

From a user perspective perhaps the most helpful improvement to the framework is the ability to visually explore all the different branches of the proof. The below image is a view of a particular leaf in an ongoing proof. The left pane shows this leaf in its position in the proof tree; the top pane shows the name of the proof “node” which is useful for closer inspection and debugging; the middle pane shows the current state in that node; and the bottom right pane what restrictions applies to the variables.



```
│  │ k: JUMPI 1252 bool2Word ( VV0__to_3c5818c8:Int ==Int 0 ) ~> #pc [ JUMPI ] ~> #execu ...
│  │ callDepth: 0
│  │ statusCode: STATUSCODE:StatusCode
│  │ pc: 1135
│  │
│  ┣━ e4e010..43fb41 (expanded)    ( ( notBool VV0__to_3c5818c8:Int ==Int 0 ) )
│  ┃  │ k: JUMPI 1252 bool2Word ( VV0__to_3c5818c8:Int ==Int 0 ) ~> #pc [ JUMPI ] ~> #execu ..
│  ┃  │ callDepth: 0
│  ┃  │ statusCode: STATUSCODE:StatusCode
│  ┃  │ pc: 1135
│  ┃  │
│  ┃  │  (189 steps)
│  ┃  └  84eba8..740a2f (frontier, leaf)
│  ┃     │ k: #halt ~> CONTINUATION:K
│  ┃     │ callDepth: 0
│  ┃     │ statusCode: EVMC_REVERT
│  ┃     │ pc: 1251
│  ┃
│  ┗━ 26379f..89f4e6 (expanded)    VV0__to_3c5818c8:Int ==Int 0
│     │ k: JUMPI 1252 bool2Word ( VV0__to_3c5818c8:Int ==Int 0 ) ~> #pc [ JUMPI ] ~> #execu ..
│     │ callDepth: 0
│     │ statusCode: STATUSCODE:StatusCode
│     │ pc: 1135
│     │
│     │  (54 steps)
│     ├  378124..5362a6 (expanded)
│     │  k: JUMPI 1285 bool2Word ( CALLER_ID:Int ==Int ORIGIN_ID:Int ) ~> #pc [ JUMPI ] ~> # .
│     │  callDepth: 0
│     │  statusCode: STATUSCODE:StatusCode
│     │  pc: 1260
│     │
│     ┣━ 38eb0c..f32d37 (frontier, leaf)    CALLER_ID:Int ==Int ORIGIN_ID:Int           ] ~> #
│     ┃   │ k: JUMPI 1285 bool2Word ( CALLER_ID:Int ==Int ORIGIN_ID:Int ) ~> #pc [ JUMPI ] ~> #
│     ┃   │ callDepth: 0
│     ┃   │ statusCode: STATUSCODE:StatusCode
│     ┃   │ pc: 1260
│     ┃
│     ┗━ 17f864..dcdfe4 (frontier, leaf)    ( ( notBool CALLER_ID:Int ==Int ORIGIN_ID:I] ~> #
│          │ k: JUMPI 1285 bool2Word ( CALLER_ID:Int ==Int ORIGIN_ID:Int ) ~> #pc [ JUMPI ] ~>
│          │ callDepth: 0
│          │ statusCode: STATUSCODE:StatusCode
│          │ pc: 1260
│
└─ 12e34a..185b1d (expanded)    VV3__isCreation_3c5818c8:Int ==Int 0
    │ k: JUMPI 1252 bool2Word ( VV3__isCreation_3c5818c8:Int ==Int 0 ) ~> #pc [ JUMPI ] ~ ...
    │ callDepth: 0
    │ statusCode: STATUSCODE:StatusCode
    │ pc: 1120
    │
    │  (54 steps)
    ├  88f95c..21f812 (expanded)
    │  k: JUMPI 1285 bool2Word ( CALLER_ID:Int ==Int ORIGIN_ID:Int ) ~> #pc [ JUMPI ] ~> # ...
    │  callDepth: 0
    │  statusCode: STATUSCODE:StatusCode
    │  pc: 1260
    │
    ┣━ a5dc95..157bca (expanded)    CALLER_ID:Int ==Int ORIGIN_ID:Int
```

```
node(17f864..dcdfe4)
```

```
<generatedTop>
  <foundry>
    <kevm>
      <k>
        JUMPI 1285 bool2Word ( CALLER_ID:Int ==Int ORIGIN_ID:Int )
        ~> #pc [ JUMPI ]
        ~> #execute
        ~> CONTINUATION:K
      </k>
      <mode>
        NORMAL
      </mode>
      <schedule>
        LONDON
      </schedule>
      <ethereum>
        <evm>
          <callStack>
            .List
          </callStack>
          <interimStates>
            .List
          </interimStates>
          <callState>
            <program>
              b"`\x80`@R`\x046\x10a\x01\x17W`\x005`\xe0\x1c\x80c\x8c1R\xe9\x11a\x00\xa0W\x80c\xcd|\x97\x89\x11a\x00dW\x80c\xcd|\x97\x89\x14a\x03 W\x80c\xcf\xf0\xab\x96\x14a\x035W\x
            </program>
            <jumpDests>
              ( ( SetItem ( 12995 ) ( ( SetItem ( 10707 ) ( ( SetItem ( 2515 ) ( ( SetItem ( 5843 ) ( ( SetItem ( 9683 ) ( ( SetItem ( 6371 ) ( ( SetItem ( 2547 ) ( ( SetItem ( 510
            </jumpDests>
            <id>
              1032069922050249630382865877677304880282300743300
            </id>
            <caller>
              CALLER_ID:Int
            </caller>
            <callData>
              ( ( b"\xe9\xe0\\B" ++ ( ( ( ( ( ( ( ( ( b"\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x
```

```
{ VV0__to_3c5818c8:Int #Equals 0 }
#rangeAddress ( 1032069922050249630382865877677304880282300743300 )
#rangeAddress ( CALLER_ID:Int )
#rangeAddress ( ORIGIN_ID:Int )
#Not ( { VV3__isCreation_3c5818c8:Int #Equals 0 } )
0 <=Int CALLER_ID:Int
0 <=Int ORIGIN_ID:Int
0 <=Int VV1__value_3c5818c8:Int
0 <=Int VV2__gasLimit_3c5818c8:Int
VV2__gasLimit_3c5818c8:Int <Int 18446744073709551616
CALLER_ID:Int <Int 1461501637330902918203684832716283019655932542976
ORIGIN_ID:Int <Int 1461501637330902918203684832716283019655932542976
```

Finally, we have found that in terms of communicating specifications both internally and with clients, writing specifications in Solidity with “cheat code” calls is simpler than building specifications in K. The common language as well as the ability to immediately compile and fuzz leads to a great reduction in specification bugs – something that is quite common when writing K specs but which we have had no serious instance of in this engagement.