A Review of the Deposit Contract

Suhabe Bugrara, ConsenSys R&D

Wednesday, April 15, 2020

Introduction

This report is a review of the deposit contract and its KEVM formal specification of the bytecode with commit hash 4290c1d on Dec 17, 2019. The only goal of the review is to identify actionable recommendations to increase confidence in the safety of the deposit contract.

Conclusion: the deposit contract bytecode is unsafe because the Vyper compiler used to compile it is unreliable. See the issue titled *Formal verification cannot be substituted for rigorous software engineering practice* below for details.

This review consisted of four steps:
1. Conduct a standard security audit of the source code of the deposit contract.
2. Confirm that the KEVM formal specification is *sound* in the sense that it encodes every important correctness property, for all possible executions of the deposit contract.
3. Make recommendations to simplify the formal specification so that it is easier to check for *soundness*.
4. Identify opportunities for future work to further increase confidence in the deposit contract

Code Audit

This section gives a list of issues found in the course of the review.

Formal verification cannot be substituted for rigorous software engineering practice

Many bugs were found in the Vyper compiler in the course of both RV's formal verification report of the deposit contract as well as Diligence's code audit. Refer to the document titled Update on the Vyper Compiler. It is concerning that the deposit contract was compiled with it. This report recommends not using the current deposit contract bytecode in production.

Formal verification cannot be substituted for rigorous software engineering practice such as using a reliable, proven, widely-used compiler with a history of robust operation. This principle is well-established in the software safety literature (See Section 12.2.2 of DO-178C: Software Considerations in Airborne Systems and Equipment Certification). Formal verification is *necessary*, but it is not sufficient to guarantee the safety of mission-critical applications.

A fundamental principle of software safety is redundancy. Formal verification at the bytecode-level is one layer of redundancy and using a reliable compiler is another layer. Each layer independently aims to achieve the same purpose, and both must be used for the mission-critical programs like the deposit contract.

Furthermore, industrial-grade formal verification systems typically consist of hundreds of thousands of lines of complex code. If the prover itself has a bug, it could invalidate the formal verification correctness result on the deposit contract. Fortunately, academic researchers have developed a cutting-edge technique called proof object generation which would address this concern, but it has not yet been incorporated into practical systems yet.

No documented testing strategy

The deposit contract repository does not include a testing strategy document for the contract, which is a basic requirement for rigorous software engineering practice. At a minimum, this document should explain why the test cases in the repository are sufficient to thoroughly test the program. It should also note the line and branch coverage they achieve.

No deployment procedures

The deposit contract repository does not include well-documented procedures for deploying the contract, which is a basic requirement for rigorous software engineering practice. Among other steps, the deployment procedures should include checking that the on-chain bytecode is correct and that storage was initialized correctly.

Unsafe handling of dead path

The `deposit` function has a dead path that is not obvious by reading the source code. This dead path iterates over the loop exactly `DEPOSIT_CONTRACT_TREE_DEPTH` times, exits the loop normally without ever having `bitwise_and(size, 1) == 1`, and then reaches the function exit.

Defensive programming dictates that an `assert false` statement should be added at the end of the function and the `break` statement should be replaced with a `return`.

Simplifying the Specification

We made the following recommendations to Runtime Verification for simplifying the bytecode formal specification so that it is easier for people to understand and check for soundness. The initial spec was at commit ac4f0bb and the final was c12cff9.

1. The constructor and `get_deposit_count` functions only have loops that are statically-sized and have simple bodies. The multiple, chained K rules for each of these functions can be replaced with a single end-to-end K rule that is simpler and easier to understand.
2. The K rules for the `get_deposit_count`, `get_deposit_root`, and `deposit` functions used a summary K rule for the call to `to_little_endian_64` which was verified separately. The value of using verified summary rules is that they can improve prover performance without increasing the trust base. However, they do increase the length of the specification. In this particular situation, we believe the succinctness of the specification is more valuable than the performance improvement.
3. The `localMem` cells of many K rules specify the value of every memory slot at loop-related program points within a function, which results in many `localMem` cells that are more than 70 lines of K which makes the rules unreadable. The specification should only specify the values of critical memory slots that affect soundness and not the ones that can be handled automatically by the K prover.
4. The `spec-impl.k` template file, which is used as base for generated K rules, includes default placeholders for 26 KEVM cells and three constraints that were not relevant for the specification.

Runtime Verification implemented and merged these recommendations. See the following pull requests:

1. PR #317
2. PR #318
3. PR #319
4. PR #320

Collectively, these simplifications had the following effects:

1. Reduced the number of K rules from 42 to 29, which is 30% fewer.
2. Reduced the total lines of the generated K rules from 11,723 to 4,112, which is 65% shorter.
3. Reduced the total lines of the INI eDSL rules from 1,430 to 800, which is 44% shorter.

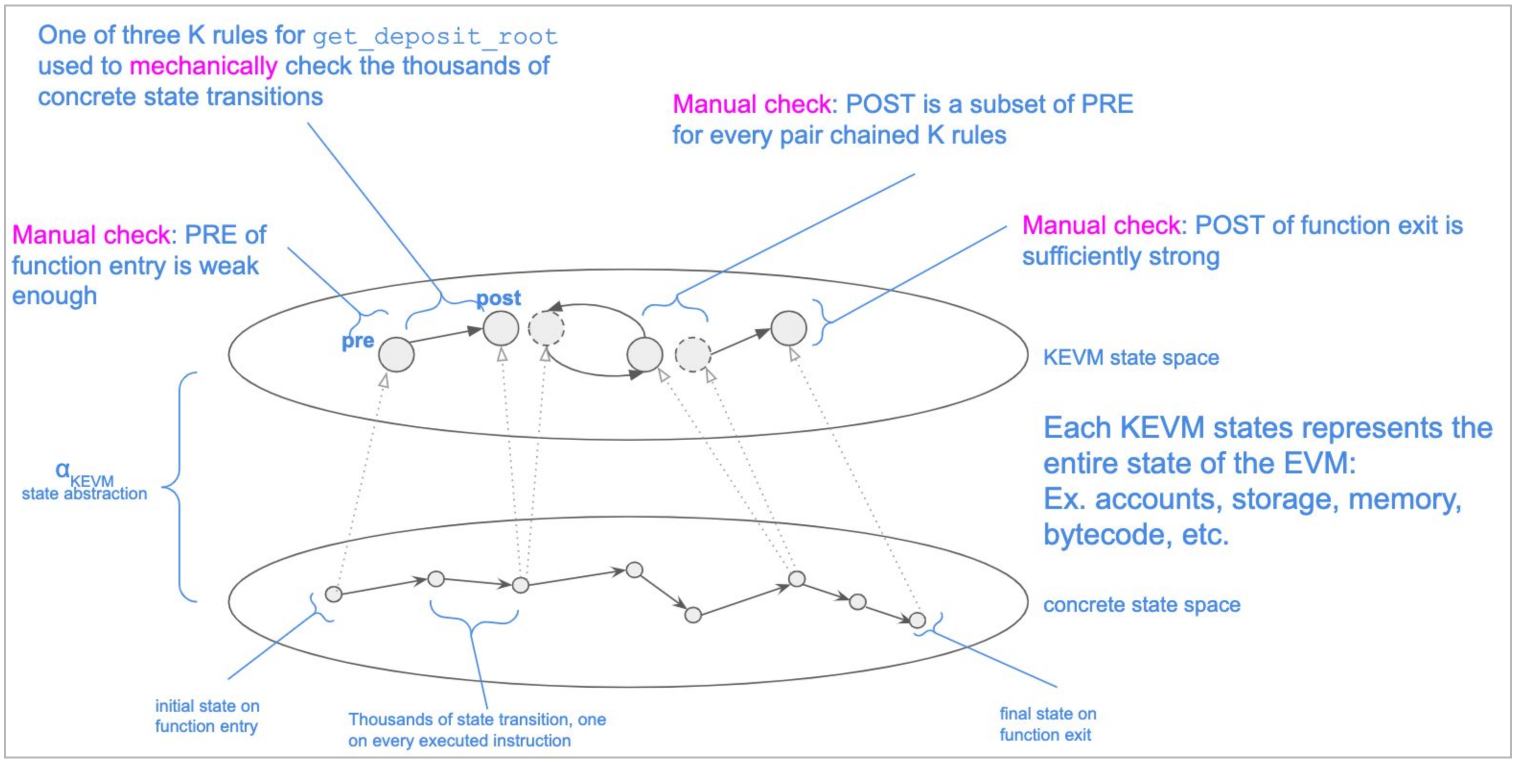
Thanks to Daejun Park for accepting and implementing these recommendations.

Checking Thoroughness of the Specification

The formal specification needed to be checked manually to ensure that it is sound in the sense that it encodes every important correctness property, for all possible executions of the deposit contract. The specification consists of a bytecode-level specification and an algorithm-level specification.

Bytecode-Level Specification

The following diagram visualizes how the bytecode-level specification for the `get_deposit_root` function is structured and identifies which parts need to be manually checked:


One of three K rules for get_deposit_root used to mechanically check the thousands of concrete state transitions
Manual check: POST is a subset of PRE for every pair chained K rules
Manual check: PRE of function entry is weak enough
Manual check: POST of function exit is sufficiently strong
post
pre
KEVM state space
Each KEVM states represents the entire state of the EVM: Ex. accounts, storage, memory, bytecode, etc.
α KEVM state abstraction
concrete state space
initial state on function entry
Thousands of state transition, one on every executed instruction
final state on function exit

The set of manual checks can be divided into the following types:

1. The preconditions of K rules that start at the entry of the contract should be collectively weak enough so as to cover all possible transactions and initial contract states.
2. The postconditions of K rules that end at the exit of the contract should be collectively strong enough so as to guarantee the correct and observable behavior of the function: storage, status code, and return value.
3. For every concrete execution of the function, there must be a sequence of K rules that capture every state of that execution at each instruction.
4. The algorithmic part of the specification ca

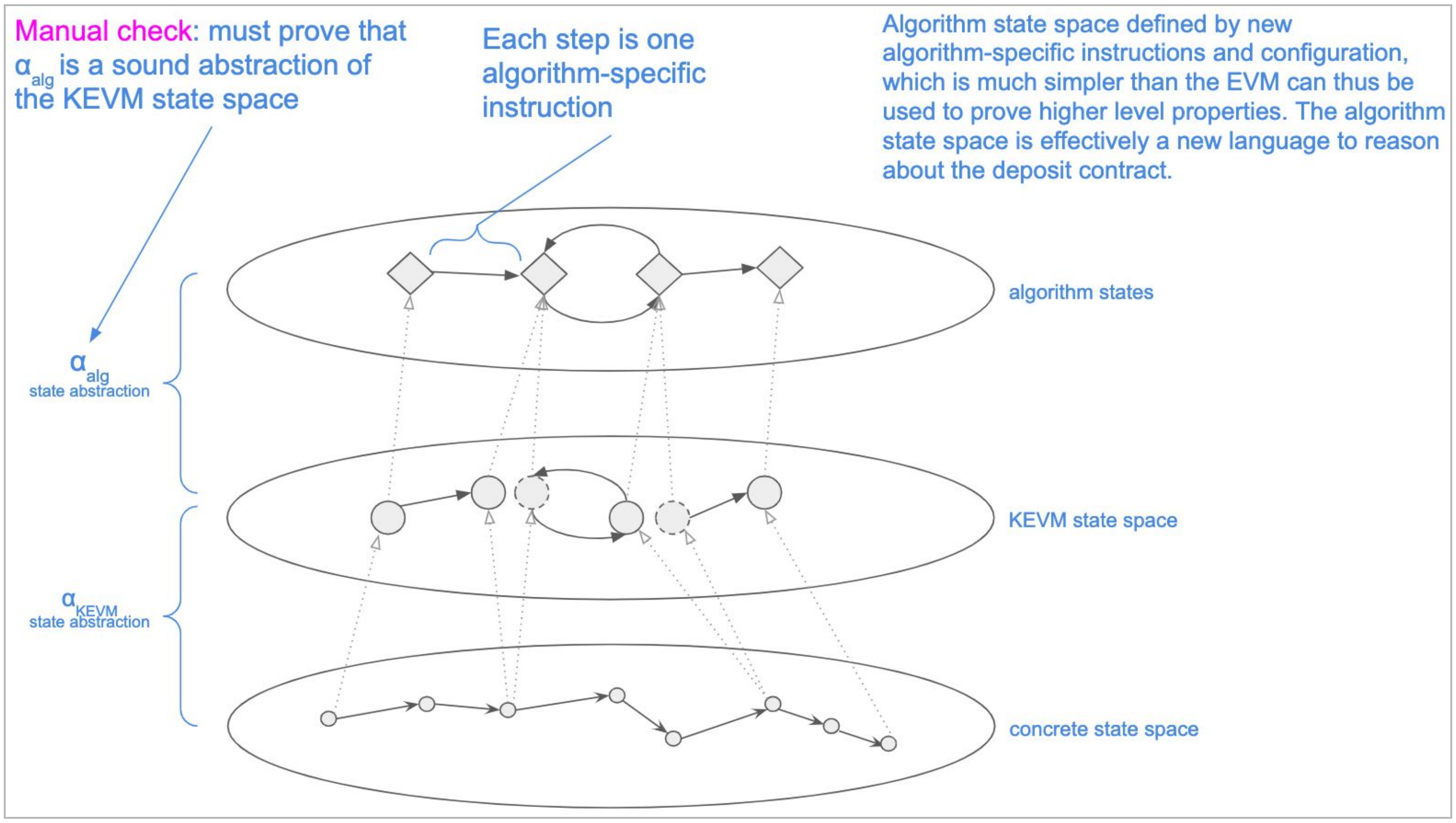
The first two types of manual checks make sure that the specification encodes the correctness properties of the function, while the third manual check is a limitation of the current implementation of the K prover, which should ideally automate it.

Underspecification

One under-specification issue was discovered in the formal specification where the `statusCode` of the constructor, `get_deposit_count`, and `get_deposit_root` functions were not being verified. Runtime Verification merged a fix in PR #316.

Algorithm-level Specification

The following diagram visualizes how the algorithm-level specification for the `get_deposit_root` function is structured. The abstraction function from the KEVM state space to the algorithm state space needs to be manually checked. Conceptually, it is possible to mechanize this step but this would require significant R&D investment into the K framework. See the article on this refinement approach and an example of how to use the `keq` tool for more details.


Manual check: must prove that α_{alg} is a sound abstraction of the KEVM state space
Each step is one algorithm-specific instruction
Algorithm state space defined by new algorithm-specific instructions and configuration, which is much simpler than the EVM can thus be used to prove higher level properties. The algorithm state space is effectively a new language to reason about the deposit contract.
algorithm states
α_{alg} state abstraction
KEVM state space
α_{KEVM} state abstraction
concrete state space

Future Work

This section gives a list of additional checks that could be performed.

1. Manually check that the formal specification encodes the liveness property that the contract is always able to accept a new, valid deposit as long as a sufficient amount of gas is provided.
2. Manually check the `gas`, `memoryUsed`, `callGas` cells of the formal specifications. These KEVM cells are the only cells not checked by this review.
3. Manually check that the formal specification correctly handles garbage calldata.
4. Remove values from the formal specification that are dependent on the specific instance of the EVM bytecode generated by the compiler. For example, the formal specification uses specific program counter values and memory addresses which would need to be updated every time the program is changed and recompiled. One of the value propositions of a formal specification is that the program can be automatically re-verified after making changes to the program without additional human effort. This work item is challenging and time-consuming, and thus it may not be worth the effort if the contract is only going to be iterated many more times.

Recommended R&D Investments

This section gives a list of recommended R&D investments that would *greatly* enhance the existing, state-of-the-art formal verification systems to be much more effective at ensuring the safety of mission-critical software like the deposit contract.

Proof Object Generation

Using a formal verifier to check the deposit contract at the bytecode level eliminates trust in the Vyper compiler, which is critical. However, you now have to trust the verifier which is typically hundreds of thousands of lines of complex code.

Proof object generation is a powerful technique that enables you not to have to trust the verifier at all.

In short, proof objects are correctness certificates. A major concern in formal verification is that verifiers do very complex things, and it becomes difficult to trust them. Many well-known verifiers have bugs. And it is almost impossible to eliminate bugs from complex software such as verifiers.

Using proof objects, we can establish the trust on a program by program basis. We do not care about the correctness of the entire verifier anymore. We only care about the correctness of each individual task that the verifier does, e.g., it verifies that a particular program P has the particular property φ.

A proof object for that task is an encoding of a detailed, rigorous, machine-checkable mathematical proof of the fact that program P has property φ. The program that checks proof objects is called a proof checker, which are programs of several hundred lines of code, so their correctness can be trusted. The correctness of the verification task (P has property φ) is then guaranteed by proof-checking the proof object.

Thanks to Xiahong Chen for helping explain proof object generation.

Certified compilation

Compilers are commonly assumed to be perfect in the sense that the compiled code is always semantically equivalent to the source program. However, compilers are highly complex programs consisting of hundreds of thousands of lines of sophisticated symbolic transformations. A single bug in the compiler can silently cause it to generate an incorrect executable. Certified compilation is a technique that applies formal methods to the compiler itself to ensure that it always preserves the semantics of the source programs it compiles.

Projects such as CompCert and CakeML show that building formally verified compilers is within reach with currently known techniques. CompCert has seen adoption for some mission-critical aerospace code at Airbus. Certified compilation can be useful even outside the context of full formal verification of software: developers who cannot justify the investment of fully verifying their program can still use a verified compiler and at least be confident that the compiler did not introduce any bugs not already present in their source code.

Thanks to Mario Alvarez for helping explain certified compilation. See Formal Certification of a Compiler Back-end for more details.

Mechanize the Checking of the Algorithm-to-Bytecode Specification Refinement

As discussed in the Algorithm-level Specification section, one of the key manual checks of the formal specification is the correctness of abstraction function from the KEVM state space to the algorithmic state space. This manual check should be fully mechanized as it is very complex thus it is very easy to get wrong.

Acknowledgements

This review was funded by ConsenSys R&D and benefited greatly from the valuable feedback given by Mario Alvarez, Joseph Chow, Robert Drost, Martin Lundfall, and Valentin Wüstholz. Thanks to Xiaohon Chen, Daejun Park, Grigore Rosu, and Yi Zhang at Runtime Verification for being extremely responsive and committed to making the deposit contract formal verification effort of the highest caliber possible.