Formal verification of Algorand smart contracts

The coq-avm library

2023-04-20

Contents

1 Introduction

1.1 The Algorand Virtual Machine and TEAL

Algorand is a blockchain platform that is designed to provide a secure, scalable, and decentralized environment for building decentralized applications. To achieve this, Algorand uses a unique consensus mechanism known as Pure Proof-of-Stake (PPoS), which ensures that all network participants have an equal opportunity to participate in block validation and decision-making processes.

The Algorand Virtual Machine (AVM) is a key component of the Algorand ecosystem. It is a software platform that allows developers to build and deploy smart contracts on the Algorand blockchain. The AVM provides a secure and sandboxed environment for executing smart contracts, ensuring that they operate in a deterministic and predictable manner. The AVM can interpret programs encoded as AVM bytecode, these programs can be implemented in PyTeal or directly in TEAL (see below), but there are plans to support other lnguages in the future.

Transaction Execution Approval Language (TEAL) is the assembly language of AVM, can be used to develop Algorand smart contracts. TEAL programs can also be written/generated using other languages and tools, such as PyTeal and Reach. During a compilation process AVM bytecode is generated, which the Algorand Virtual Machine can directly interpret during smart contract execution/validation in transactions.

Creating secure smart contracts is essential because they are the backbone of decentralized applications and blockchain technology. Smart contracts are programs that run on a blockchain, and they facilitate transactions and agreements between parties without the need for intermediaries. Because smart contracts are immutable and enforceable, they can be used to automate complex business processes, such as supply chain management, financial transactions, and voting systems. However, if smart contracts are not secure, they can be vulnerable to hacks, exploits, and other security threats, which can result in the loss of funds, data breaches, and other serious consequences. To avoid these risks, it is crucial to create secure smart contracts that are designed to be resilient to attacks and adhere to best practices for security and safety.

1.2 Coq

Coq is an interactive theorem prover and proof assistant, used for formal verification of mathematical proofs, algorithms, and software programs. It is an open-source tool that uses the calculus of inductive constructions to support formal proofs, making it a reliable and powerful tool for the verification of complex systems. Coq allows users to define mathematical functions and data structures, as well as to prove properties and theorems about them, using a rich type system and a high-level functional programming language.

Coq can be used to formally verify smart contracts, ensuring their correctness and security. Smart contracts are programs that run on a blockchain and are designed to automate complex business processes. However, because smart contracts operate in a decentralized and trustless environment, they are vulnerable to security threats, such as hacks, exploits, and bugs. To mitigate these risks, formal verification can be used to prove the correctness and safety of smart contracts. Coq can be used to create

mathematical models of smart contracts and to verify their properties and behavior, ensuring that they function as intended and are free of vulnerabilities. Formal verification using Coq can help to increase the trust and reliability of smart contracts, making them more secure and effective in a wide range of applications.

Coq, however does not support AVM bytecode out of the box, this is where the coq-avm library is needed. The library aims to be a highly reliable implementation of the AVM interpreter (along with formal proofs that help developments), and as such can be used as an "executable" formal model. This approach allows writing program proofs much easier (as opposed to axiomatic or operational semantics), and performing simple evaluations of compiled TEAL programs are as easy as invoking the `compute` tactic of Coq.

2 Benefits of using formal verification tools

2.1 Benefits for Algorand developers

The original Algorand Virtual Machine is implemented in the Go programming language and while great efforts are taken on documenting the individual op codes, not all behaviour can be extrapolated from the docs. Having a formal specification on the effects of op codes could help documentation efforts on the AVM and TEAL.

Furthermore with the help of the library efforts could be made to construct proofs that Algorand apps and smart signatures behave the same way on the next version of the AVM.

Since Coq also supports code extractions to OCaml, Scheme and Haskell languages natively, additional implementations can be generated based on the library.

2.2 Benefits for smart contract validators

As of now, there is a formal verification tool in beta version for Algorand programs implemented for the K framework, additional tools could help construct formal proofs for different scenarios.

2.3 Benefits for smart contract developers

Needless to say, proving properties of Algorand programs should incite a higher confidence than just running a manually constructed test suite. Coq, on the other case is also capable as a modelling tool, verifying interactions between different Algorand applications or proving much needed invariants on global/local state of the apps.

Futhermore, coq-avm implements a number of tactics to help in step-by-step evaluation of the bytecode, "debugging" can be much easier, and helps understanding some bugs, by examining the state of the interpreter after each step.

2.4 Benefits for users of smart contracts

While the AVM bytecode for programs is available on chain, it cannot be expected from most users to have the technical knowledge to be able to use Coq to evaluate the outcome of specific applications they intend to use (however some users may have the expertise to do that). Proved Coq properties of applications in theory could be shared online with the appropriate proof scripts the users can verify by executing the compilation steps.

3 The current state of the coq-avm library

The coq-avm library currently implements the interpretation of 25 op codes out of the 173 op codes available in version 8. The implementation is not checked against the original AVM implementation yet.

These op codes implementations are, however enough to state and prove some properties of parts of the example program "auction-demo". The part which the proofs are focused on is the first section of the smart contract, the creation. While this execution path is very "basic", it is still a useful showcase, as it contains decisions, conversions and checking the correctness of global state values during the program execution. The part under our investigation looks like this (in Python):

```
... # code omitted
on_create_start_time = Btoi(Txn.application_args[2])
on_create_end_time = Btoi(Txn.application_args[3])
on_create = Seq(
  App.globalPut(seller_key, Txn.application_args[0]),
  App.globalPut(nft_id_key, Btoi(Txn.application_args[1])),
  App.globalPut(start_time_key, on_create_start_time),
  App.globalPut(end_time_key, on_create_end_time),
  App.globalPut(reserve_amount_key, Btoi(Txn.application_args[4])),
  App.globalPut(min_bid_increment_key, Btoi(Txn.application_args[5])),
  App.globalPut(lead_bid_account_key, Global.zero_address()),
  Assert(
      And(
          Global.latest_timestamp() < on_create_start_time,
          on_create_start_time < on_create_end_time,
      )
  ),
  Approve(),
)
```

Listing 1: auction-demo on-create in PyTeal

Compiling the original auction-demo code with version 8 of PyTeal yields the following TEAL program:

```
#pragma version 8
txn ApplicationID
int 0
==
bnz main_l25
...
main_l25:
byte "seller"
txna ApplicationArgs 0
app_global_put
byte "nft_id"
txna ApplicationArgs 1
btoi
app_global_put
byte "start"
txna ApplicationArgs 2
btoi
app_global_put
byte "end"
txna ApplicationArgs 3
btoi
app_global_put
byte "reserve_amount"
txna ApplicationArgs 4
btoi
app_global_put
byte "min_bid_inc"
txna ApplicationArgs 5
btoi
app_global_put
byte "bid_account"
global ZeroAddress
app_global_put
global LatestTimestamp
txna ApplicationArgs 2
btoi
<
txna ApplicationArgs 2
btoi
txna ApplicationArgs 3
btoi
<
&&
assert
int 1
return
```

Listing 2: auction-demo on-create in TEAL

This TEAL program is compiled to AVM bytecode, which is imported into the Coq environment as `ApprovalProgram.programByteCode`. Here is a simple execution of the program with the library:

```
Property approvalProgram_test1 :

   let context :=
     startingContext [ [x00] (* seller *)
                     ; [x01] (* nftId *)
                     ; [x02] (* startTime *)
                     ; [x03] (* endTime *)
                     ; [x04] (* reserveAmount *)
                     ; [x05] (* minBidIncrement *)
                     ] in
   exists finalState,
   executeProgram context 1000 ApprovalProgram.programByteCode = (success stop, finalState).
Proof.
  compute.
  now eexists.
Qed.
```

Listing 3: auction-demo simple on-create execution with coq-avm

For a more elaborate example, here is a property that can be proved with the help of the library:

```
1  Property approvalProgram_check :
2
3    forall context seller nftId startTime endTime reserveAmount minBidIncrement,
4      latestTimestampValue (global context) < startTime →
5      startTime < endTime →
6      context = startingContext [seller; to_bytes nftId; to_bytes startTime;
7                 to_bytes endTime; to_bytes reserveAmount; to_bytes minBidIncrement] →
8
9    exists finalState,
10     executeProgram context 1000 ApprovalProgram.programByteCode = (success stop, finalState)
11     ∧ getGlobalAppStateValue "seller" finalState = Some (Bytes seller)
12     ∧ getGlobalAppStateValue "nft_id" finalState = Some (Int nftId)
13     ∧ getGlobalAppStateValue "start" finalState = Some (Int startTime)
14     ∧ getGlobalAppStateValue "end" finalState = Some (Int endTime)
15     ∧ getGlobalAppStateValue "reserve_amount" finalState = Some (Int reserveAmount)
16     ∧ getGlobalAppStateValue "min_bid_inc" finalState = Some (Int minBidIncrement)
17     ∧ getGlobalAppStateValue "bid_account" finalState
18           = Some (Bytes (zeroAddressValue (global context))).
19
20 Proof.
21   (* proof omitted *)
22 Qed.
```

Listing 4: auction-demo on-create property with coq-avm

The proof of this property is omitted due to its more complicated nature. However it proves that choosing any application arguments (line 3), such that:

- `startTime` is after the global `latestTimestamp` value (line 4)
- `startTime` is before `endTime` (line 5)
- and the context corresponds to a starting context of the app (with applicationID set to 0) (line 6-7)

will result in a successful execution (line 10) and the application global state is set according to the values chosen originally (lines 11-18).

Note that since the implementation is not verified/checked against the original AVM implementation (yet), the actual execution/proof may not entirely match the execution of the same program if ran on the Algorand blockchain.

4 Summary

The aim of coq-avm library is to help developing and understanding the semantics of the TEAL assembly language and any compiled AVM program. The project even in its early stage shows great promise in development of proofs for properties of smart contracts, ensuring their quality and security.

At its current stage the license of library is not determined yet, so a code listing is not provided; the author is available to showcase the library, however.