Defining the Ethereum Virtual Machine for Interactive Theorem Provers

Yoichi Hirai

Ethereum Foundation

Workshop on Trusted Smart Contracts
Malta, Apr. 7, 2017

Outline

Outline

1. Overview
   - Why Prove Ethereum Programs Correct
   - We Defined EVM for Theorem Provers
2. Some Technicality
   - EVM
   - Choice on Reentrancy
3. Own Evaluation
   - Remaining Problems
4. Summary

Ethereum: Public Ledger with Code

Public ledger with accounts:

- . . . some controlled by private key holders,
- . . . the others (called Ethereum contracts) controlled by code stored on the ledger.

Accounts (including Ethereum contracts) can call other accounts and send balance.

Calls invoke code in Ethereum contracts.

Bugs in Ethereum Programs.

- The DAO: funds moved much more than expected / led to network split into two
- Programs stop working when array iteration becomes too long
- Ethereum Name Service (prev. version): in a secret auction, bids could be added after other bids were revealed

  ⋮

This does not work:

1. Develop the source code of Ethereum contracts on GitHub.
2. Enough people would look at it.
3. Bugs would be found early enough.

Potential Ways to Prevent Bugs in Ethereum Programs.

Testing can check prepared scenarios
cannot find unknown attacks without luck

Code review sometimes finds attacks
Never known: how much review is enough?

Machine-checked theorem proving can enumerate everything that can happen, if it finishes.
You can see when proofs finish.

Why Formal Proofs might Make Sense for Ethereum Contracts

My speculation: for Ethereum contracts the benefit of proving might outweigh the costs.

- You cannot change deployed programs
  - Bugs remain.
  - An upgradable Ethereum contract is somehow at odds with the cause of decentralization.
- The bugs are visible to all potential attackers
- Ethereum contracts sometimes manage big amount of fund

Need of a Definition of a Programming Language in Theorem Provers

In some cases, the semantics looks like an interpreter.
In other cases, it contains clauses of possibilities.

- The definition in theorem provers is code,
  but it should be readable/comparable against spec.
- The definition needs to be tested
  - Goal: what happens on-chain should be an instantiation of the definition in theorem provers

Outline

We Defined the Ethereum Virtual Machine for Isabelle/HOL, HOL4 and Coq

- Coq (27 yrs. old), Isabelle (31 yrs. old) and HOL4 (ca. 30 yrs. old) are interactive theorem provers, where
  - one can develop math proofs and have them checked.
  - one can also develop software and prove correctness.
- “Programs” look similar in all these theorem provers

Strategic Goal: inviting users of these tools to Ethereum contract verification.

Our EVM Definition is Originally in Lem

We used a language called Lem.

- Lem code can be translated into HOL4, Isabelle/HOL, Coq and OCaml.

How the paper spec and Lem spec look

The EVM definition in Lem has 2,000 lines.
Most instructions are simply encoded as functions in Lem...
Yellow Paper (original spec):

0x06	MOD	2	1	Modulo remainder operation. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \begin{cases} 0 & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[1] = 0 \\ \boldsymbol{\mu}_{\mathbf{s}}[0] \bmod \boldsymbol{\mu}_{\mathbf{s}}[1] & \text{otherwise} \end{cases}$

Lem:

```
| Arith MOD -> stack_2_1_op v c
    (fun a divisor -> (if divisor = 0 then 0 else
        word256FromInteger ((uint a) mod (uint divisor))
    ))
```

...except CALL and friends.

Special Treatment of CALL

During CALL instruction, nested calls can enter our program.

Nasty effects after executing CALL:

- the balance of the contract might have changed
- the storage of the contract might have changed

Our blackbox treatment of CALL:

- by default, the storage and the balance change arbitrarily during a CALL.
- optionally, you can impose an invariant of the contract, which is assumed to be kept during a CALL
  but you are supposed to prove the invariant.

Currently, we are working on a precise model of what happens during a CALL.

We Tested Our EVM Definition against Implementations' Common Test

- Luckily, we have test suites for EVM definitions
  - The test suites compare Ethereum Virtual Machine implementations in Python, Go, Rust, C++, . . .
  - All EVM implementations need to behave the same, lest the Ethereum network forks (ugly)
- Definitions in Lem are translated into OCaml
- Our OCaml test harness reads test cases from Json, runs the Lem-defined EVM, checks the result v.s. expectations in Json
- VM Test suite: 40,617 cases (24 cases skipped; they involve multiple calls)
  Running those 24 involves implementing multiple calls (current efforts).

Problems in LaTeX Specification

Test suits are the spec in effect; the LaTeX spec is not tested.

While writing definitions in Lem (or previously in Coq)

- memory usage when accessing addresses $[2^{256} - 31, 1)$
- an instruction had a wrong number of arguments
- ambiguities in signed modulo:
  $\mathbf{sgn}(\mu_\mathbf{s}[0])|\mu_\mathbf{s}[0]| \bmod |\mu_\mathbf{s}[1]|$
- some instructions touched memory but did not charge for memory usage
- malformed definition: **o** was defined to be **o**

While testing the Lem definition:

- spurious modulo 2^{256} in read positions of call data
- exceptional halting did not consume all remaining gas

Proving Theorems about Ethereum Programs

We used Isabelle/HOL to prove theorems about Ethereum programs.
One theorem about a program (501 instructions) says:

- If the caller's address is not at the storage index 1, the call cannot decrease the balance
- On the same condition, the call cannot change the storage

Techniques:
Brute-force directly on the big-step semantics
(naïvely ignoring many techniques from 1960's and on).

- Human spends 3 days constructing the proof
- Machine spends 3 hours checking the proof

An Invariant

Well-defined, but questionable as documentation.

```
inductive fail_on_reentrance_invariant :: "account_state ⇒ bool"
where
 depth_zero:
  "account_address st = fail_on_reentrance_address ⟹
   account_storage st 0 = 0 ⟹
   account_code st = program_of_lst
    fail_on_reentrance_program program_content_of_lst ⟹
   account_ongoing_calls st = [] ⟹ account_killed st = False ⟹
   fail_on_reentrance_invariant st"
| depth_one:
  "account_code st = program_of_lst
    fail_on_reentrance_program program_content_of_lst ⟹
   account_storage st 0 = 1 ⟹
   account_address st = fail_on_reentrance_address ⟹
   account_ongoing_calls st = [(ve, 0, 0)] ⟹
```

Outline

Overall Data Structure

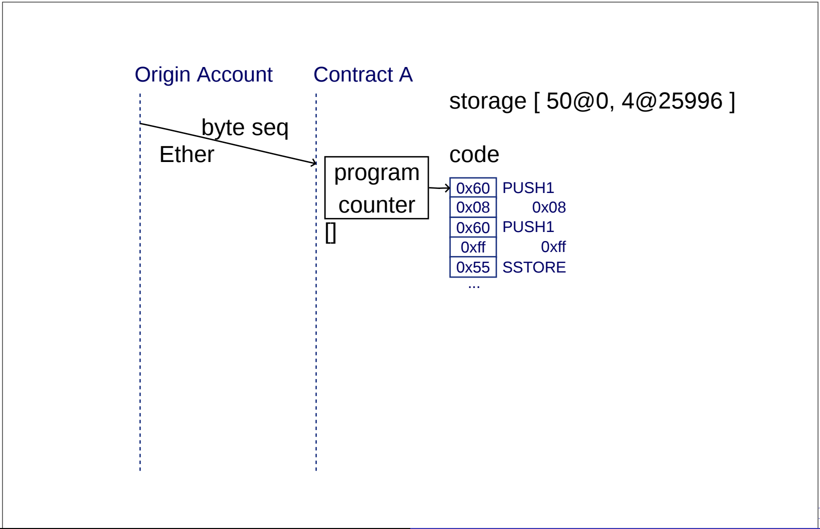
An account contains:

- balance (256-bit word)
- code (byte sequence)
- storage (2^{256} words)
- nonce (256-bit word)

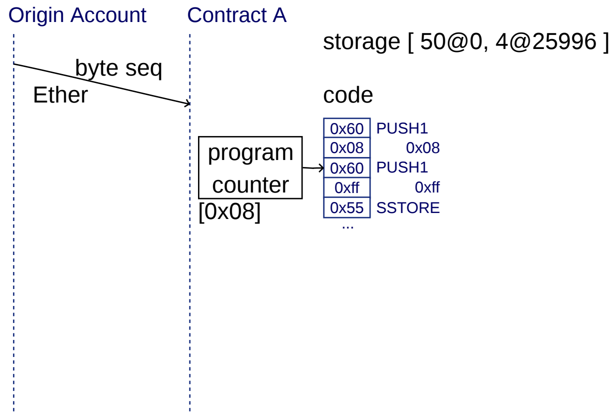
A contract invocation provides:

- input data (byte sequence)
- memory (2^{256} bytes, charged by max accessed word)
- stack (up to 1024 words)
- information by miner (timestamp, block number etc)

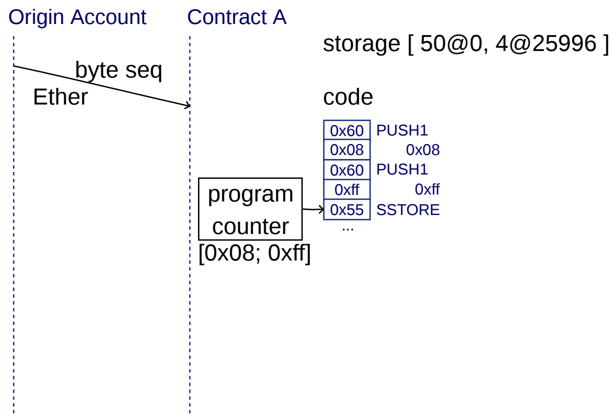
How EVM Works 1



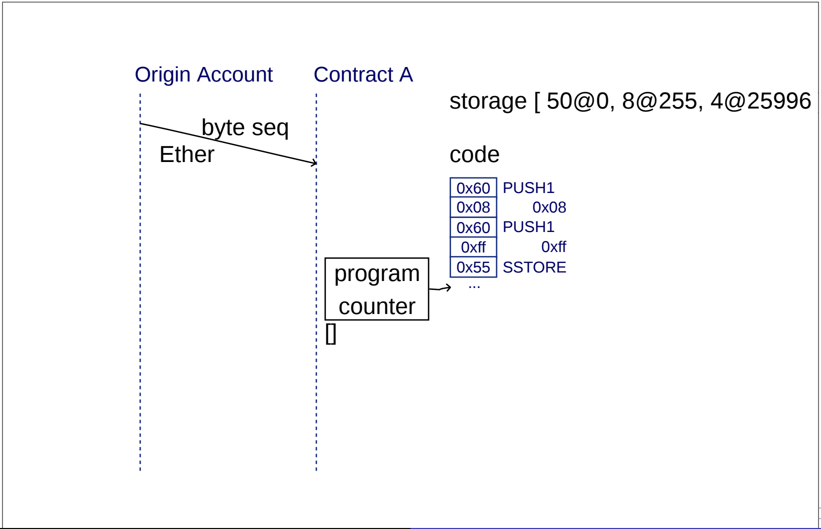
How EVM Works 2



How EVM Works 3

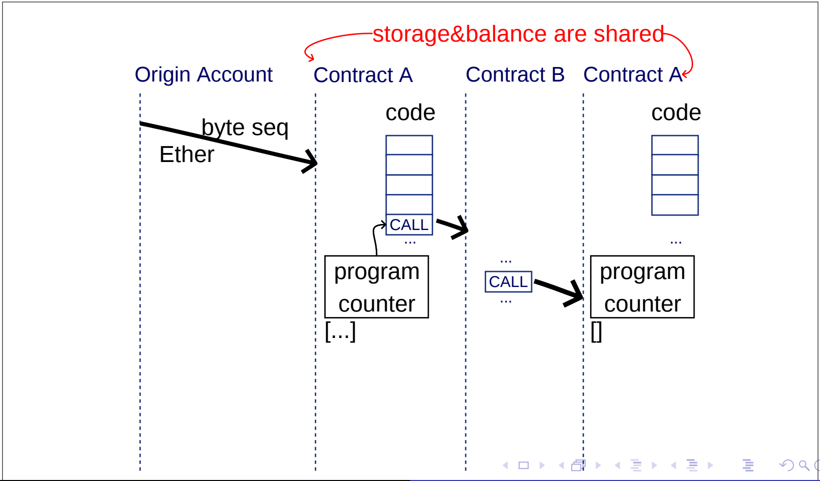


How EVM Works 4

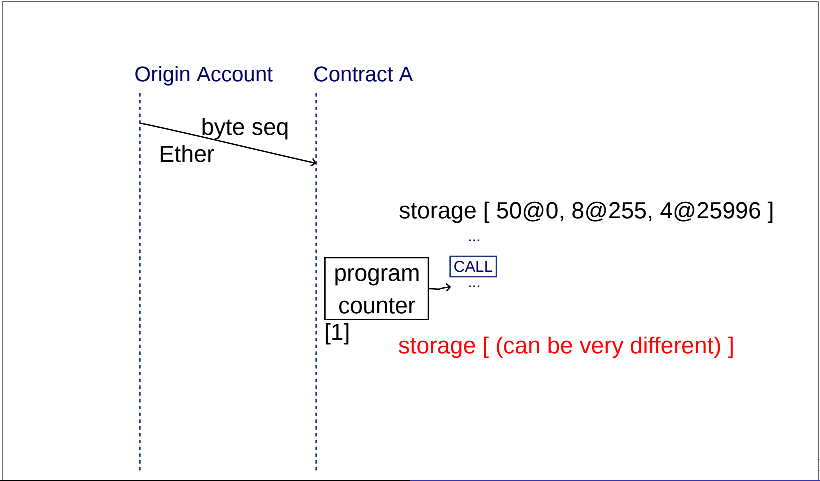


Outline

An Annoying Phenomenon Called Reentrancy (transaction's view)



An Annoying Phenomenon Called Reentrancy (invocation's view)



We Picked the Invocation's View

Pro

- A partial implementation of the other approach
- Just enough for program syntax, no bigger view necessary

Con

- Unnecessary diversion from the implementations/spec
- Complexity due to mixture of determinism/nondeterminism

After the paper. . .

We got a deterministic definition that covers a whole block (now some newly-covered tests are failing).

One Proving Strategy that We Took

1. Speculate an invariant of a contract
   "the code of the account can only stay the same or become empty"
2. Prove the invariant, assuming the invariant on reentrant calls
3. (hand-waiving argument that reentrant depth is finite)
4. Take the invariant for granted and prove pre-post conditions
   "if the caller is not the owner, the balance of the account does not decrease"

Outline

1. Overview
   - Why Prove Ethereum Programs Correct
   - We Defined EVM for Theorem Provers
2. Some Technicality
   - EVM
   - Choice on Reentrancy
3. Own Evaluation
   - Remaining Problems
4. Summary

What can still Go Wrong

This work only connects EVM spec and programs' properties

Things can go wrong with/above programs' properties

- Proven properties are different from desired ones.
- Signature forged / inverse of hash functions computed.
- An exchanges calls Ethereum contracts on behalf of users with wrong parameters (as reported yesterday)

Things can go wrong with/below EVM spec

- Bug in EVM definition can turn the theorems valueless.
- Protocol changes.

Theorem provers have bugs sometimes

More Work

Ongoing:

- definition of a whole block, containing transactions containing calls
- modular reasoning on bytecode snippets (Hoare logic w/ separating conjunction)

Not started:

- common Ethereum contract method/argument encoding
- specification language for end-users of smart contracts
- connect to test/main network

Summary

- We defined EVM for proof assistants Isabelle/HOL, Coq and HOL4
- The EVM definition is usable for proving Ethereum contracts correct for a specification

- Outlook
  - Formalization efforts underway for multiple message calls
  - Proof/tool/language/protocol developments in the proof assistants welcome
    `https://github.com/pirapira/eth-isabelle`
    (Apache License ver. 2)