Defining the Ethereum Virtual Machine for Interactive Theorem Provers

Yoichi Hirai

Ethereum Foundation `yoichi@ethereum.org`

Abstract. Smart contracts in Ethereum are executed by the Ethereum Virtual Machine (EVM). We defined EVM in Lem, a language that can be compiled for a few interactive theorem provers. We tested our definition against a standard test suite for Ethereum implementations. Using our definition, we proved some safety properties of Ethereum smart contracts in an interactive theorem prover Isabelle/HOL. To our knowledge, ours is the first formal EVM definition for smart contract verification that implements all instructions. Our definition can serve as a basis for further analysis and generation of Ethereum smart contracts.

1 Introduction

Ethereum is a protocol for executing a virtual computer in an open and distributed manner. This virtual computer is called the *Ethereum Virtual Machine* (EVM). The programs on EVM are called Ethereum *smart contracts*. A deployed Ethereum smart contract is public under adversarial scrutiny, and the code is not updatable. Most applications (auctions, prediction markets, identity/reputation management etc.) involve smart contracts managing funds or authenticating external entities. In this environment, the code should be trustworthy.

The developers and the users of smart contracts should be able to check the properties of the smart contracts with widely available proof checkers. Our EVM definition is written in Lem, which can be translated into popular interactive theorem provers Coq [1], Isabelle/HOL [19] and HOL4 [23]. We used our EVM definition and proved safety properties of some smart contracts in Isabelle/HOL.

Our contributions are as follows:

- we gave a formal specification of the interface between a smart contract execution and the rest of the world (Sec. 4);
- we defined EVM in a way portable to different interactive theorem provers (Isabelle/HOL and HOL4) and a programming language OCaml, during which we found some subtle differences between the specification (the Yellow Paper [26]) and the implementations (Sections 5 and 6);
- we tested the executable part of our EVM definition against the VM test suite, which validates existing Ethereum node implementations (Subsection 5.3); we found unsearched corner cases in the test suite;
- we used our EVM definition to prove invariants and safety properties of Ethereum smart contracts (Sec. 7).

2 Choice of the Goal and the Tool

2.1 Goal: Which Programming Language to Formalize

Considerations around Solidity Although ultimately all Ethereum smart contracts are deployed as EVM bytecode, the bytecode is rarely directly written. The most popular programming language Solidity [3] has a rich syntax but no specification. The only definition of Solidity is the Solidity compiler implementation, which compiles Solidity programs into EVM bytecode.

The Solidity compiler is written in C++. Importing the C++ code in a theorem prover is nearly impossible because the definition of the whole C++11 language has not been formalized although some of the hardest aspects of the language have been addressed: concurrency [6], inheritance [21] etc.

It is feasible to verify a compiler with optimization (e.g. CompCert [14] and CakeML [13]). Something similar for Solidity would require formalization of both Solidity and EVM before correctness of the compiler can be stated.

Considerations on EVM There are drawbacks of verifying EVM bytecode:

– most developers and users do not read EVM bytecode;
– the EVM architecture might become obsolete after the protocol adopts one of the proposed new architectures (EVM 1.5 that introduces function calls or EVM 2.0 which is based on WebAssembly [4]).

The first point can be, in the future, mitigated by translating static assertions in Solidity into EVM bytecode. The second point is, in fact, milder compared with the fast changes of the Solidity compiler. When the virtual machine architecture changes, all Ethereum implementations need to implement the change. This makes EVM change slower than the Solidity compiler.

EVM is an attractive formalization target. It is a stack-machine with a simple instruction-encoding and fully sequential execution. The simplicity of the EVM architecture resulted in just over 2,000 lines of formal definition. EVM has an English specification called the Yellow Paper (Fig. 1) clear enough to allow multiple implementations to be developed independently[1]. Also, since any disagreements among implementations hurt the availability of the network, the community has implemented test suites to compare EVM implementations. We use one of these test suites to test our EVM definition.

2.2 Tool: Formalization in Which Language

We intend our EVM definition as a basis for smart contract verification. The verification should be done in a precise manner. Model checkers are not capable of doing this because they cannot treat the huge state space: a smart contract can store up to 2^{256} 256-bit machine words permanently (the resource usage is limited

[1] Several entities develop Ethereum clients in Python, C++, Rust, Java, Scala and Go, and each contains its own EVM implementation.

0s: Stop and Arithmetic Operations

All arithmetic is modulo 2^{256} unless otherwise noted.

Value	Mnemonic	δ	α	Description
0x01	ADD	2	1	Addition operation. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\mu}_{\mathbf{s}}[0] + \boldsymbol{\mu}_{\mathbf{s}}[1]$
$\vdots$				
0x08	ADDMOD	3	1	Modulo addition operation. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \begin{cases} 0 & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[2] = 0 \\ (\boldsymbol{\mu}_{\mathbf{s}}[0] + \boldsymbol{\mu}_{\mathbf{s}}[1]) \mod \boldsymbol{\mu}_{\mathbf{s}}[2] & \text{otherwise} \end{cases}$ All intermediate calculations of this operation are not subject to the 2^{256} modulo.

Fig. 1. A short excerpt from the Yellow Paper [26]. The symbol δ (resp. α) stands for the number of deleted (resp. added) stack elements. $\boldsymbol{\mu}_{\mathbf{s}}[i]$ is the i-th stack element before the instruction execution. $\boldsymbol{\mu}'_{\mathbf{s}}[i]$ is the i-th stack element afterwards.

only economically). Such big state spaces can better be dealt with interactive theorem provers. Instead of specifying EVM in one particular theorem prover, we chose a framework called Lem [16] because definitions in Lem can be translated into some popular theorem provers: Coq [1], Isabelle/HOL [19] and HOL4 [23].

One potential alternative is the K-Framework [22]. The K-Framework is a tool specifically engineered for defining programming languages. We chose Lem and its translation targets for their larger user base[2] and their longer history.

3 A Brief Description of the Ethereum Virtual Machine

Some of our design choices and challenges can be described only after an overview of EVM. We just describe EVM as a state machine that executes programs. We omit the underlying techniques that support distributed execution.

3.1 States

In EVM, apart from several global parameters, most states are stored in accounts. EVM has a partial map from *addresses* (160-bit words) to account states. An *account state* contains code, storage, nonce and the balance. The *code* is a sequence of bytes. The *storage* is a mapping from a machine word (an EVM *machine word* has 256 bits) to a machine word. The *nonce* is an ever-increasing machine word. The *balance* is also a machine word, representing some transferable value that can be paid as fees to run EVM. When the code is not empty,

[2] The Coq users' mailing list has 1,404 subscribers while the K-Framework's has 127 at the time of writing.

the code controls the account; such an account is called a *contract*. Otherwise, the account is controlled by the holder of a private key corresponding to the address; such an account is called an *external account*. The code, when exists, encodes a sequence of instructions. Instructions are all encoded in a single byte except for the PUSH instructions, which contain immediate values.

3.2 State Transitions

An external account can initiate a *transaction* either by creating a contract or by calling an account. Once a transaction is initiated, the whole state transition of EVM is deterministic. We do not describe the contract creation by an external account because a contract's state after creation is publically checkable.

Both external accounts and contracts can *call* an account. When an account calls an account, the call is accompanied with transferred balance, gas, and data. The transferred balance is deposited to the called account. The gas regulates the resource consumption during the call. When the called account is an external account, a simple balance transfer happens. Otherwise, when the called account is a contract, after the balance transfer, the called contract's code is executed. The code execution can alter the storage of the executing account. The execution can read all accounts' balances and codes.

The resource consumption of the code execution is capped by the amount of gas that the initiating external account pays for. Executing an instruction consumes some amount of gas. When the gas is exhausted, the execution fails (*out-of-gas failure*). Such failures revert all state changes performed during the current call, except gas consumption.

A contract can call an account by executing the `CALL` instruction. The ensuing balance transfer and code execution belong to the same transaction as the calling code execution. The calling contract can limit the resource consumption in the called contract by choosing the amount of gas passed on. When the inner call fails, the side-effects of the inner call is reverted (except gas consumption) but the side-effects of the outer call remains intact. The outer call is informed of such a failure through a return value of the `CALL` instruction.

A transaction belongs to a block. A *block* is a unit of agreement among Ethereum nodes. EVM has special instructions that reads the *block number* and the cryptgraphic hash values of some previous blocks. Since a block specifies a previous block but not a unique successor, blocks in the network form a tree in general, but, as far as the states of EVM are concerned, only one branch in the tree matters. Because of this, we can think of EVM as a sequencially executed machine.

4 Interface of a Contract Invocation

4.1 Boundary between the System and the Environment

We are interested in propositions of the form: whatever the environment does, the system responds in a desired manner. Before we try to specify the desired behavior, we need to identify the system and the environment. The choice is not straightforward because multiple parties are involved in EVM.

One way is to say that the system is the contract. In that case, the environment contains anything out of EVM and all accounts on EVM except the contract under verification. In our development, the system is a single contract invocation, which is even narrower than a single account (Fig. 2 (b)). The difference can be seen in the following scenario. The environment can call into the contract. The contract can reply by calling an account. The environment can, depending on the states of accounts that we do not control, call our contract again. This is called *reentrancy*. During reentrancy, the storage and the balance of our contract might change. We chose to model the reentrancy as part of the environment. We explain this choice in Subsection 6.3.

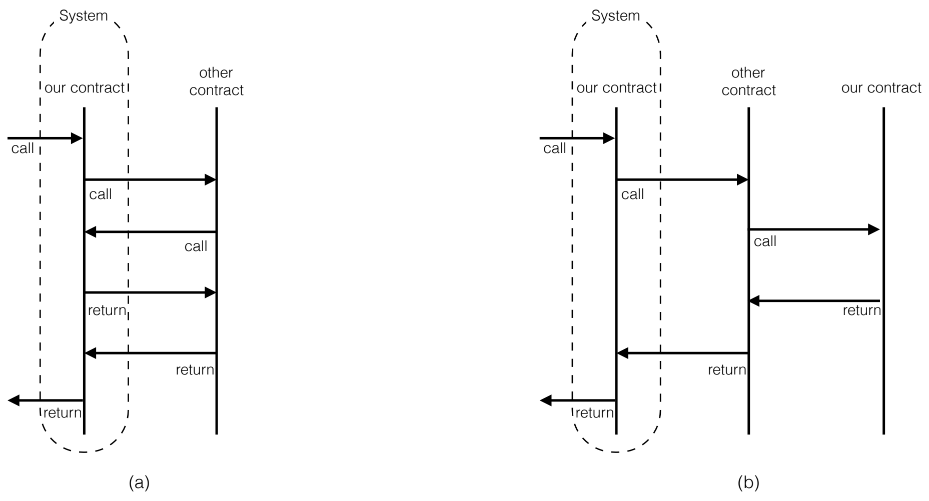

Fig. 2. Different choices of system-environment boundaries during reentrancy. Both pictures describe the same situation, but have different boundaries between the system and the environment. (a) When the system is our contract, the reentrant call is a part of the system. (b) When the system is a single invocation of our contract, the reentrant call is a part of the environment. Both are sound, but we chose (b) because it matches the program sytax where `CALL` instructions are followed by the next operations in the same message call, not the next operations in the reentrant call.

4.2 Input and Output of a Deployed Ethereum Smart Contract

In Subsection 4.1, we have set the boundary between the smart contract and the environment. Next, we identify their interaction. The specification of the interface is particularly important because it can be used to specify higher level languages for Ethereum smart contracts. Our most concrete contribution is our

EVM definition in Lem, so we show some snippets in this section and explain the syntax.

The interaction between the contract and the environment always starts with the envrionment's call into the contract. The environment can call into the contract with the following information:

```
type CALL_ENV = ⦇
 callenv_gaslimit : W256;          (* the current invocation's gas limit *)
 callenv_value : W256;                  (* the amount of Eth sent along*)
 callenv_data : LIST BYTE;                        (* the data sent along *)
 callenv_caller : ADDRESS;                      (* the caller's address *)
 callenv_timestamp : W256;       (* the timestamp of the current block *)
 callenv_blocknum : W256;     (* the block number of the current block *)
 callenv_balance : ADDRESS → W256; (* the balances of all accounts. *) ⦈
```

The whole syntax defines a *record* type with seven *fields*. A value of CALL_ENV consists of seven values each accessible under a field name. The field names are italicized. Each field name is annotated with a type of the associated value. W_{256} denotes the type of 256-bit machine words and ADDRESS 160-bit machine words. LIST BYTE is the type of lists of bytes. The arrow type ADDRESS $\rightarrow$ W_{256} is the type of total functions that take an ADDRESS and return a W_{256}. This definition is useful not only for reasoning about EVM bytecodes but also for desining high level languages that would be compiled into EVM. Ethereum contracts written in any language needs to take the combination of data above.

The environment can also make a called account return or fail after our contract makes a call. Together, the environment's possible actions are described by the following *variant type* ENVIRONMENT_ACTION, whose value can be the label EnvironmentCall together with a value of CALL_ENV, the label EnvironmentRet together with a value of RETURN_RESULT, or the label EnvironmentFail. It is automatically understood that values with different labels are different. This definition describes everything that can happen to an Ethereum contract. If we have checked these cases, we have enumerated all possibilities.

```
type ENVIRONMENT_ACTION =
| EnvironmentCall of CALL_ENV (* the environment calls the contract *)
| EnvironmentRet of RETURN_RESULT        (* the environment returns *)
| EnvironmentFail                          (* the environment fails *)
```

We omit the definition of RETURN_RESULT and many other symbols. The whole formalization is publicly available[3].

The contract can also make its move: calling another account, making a delegate call, creating a contract, failing, destroying itself, or returning. A *delegate call* runs a potentially different account's code on the caller's account.

```
type CONTRACT_ACTION =
| ContractCall of CALL_ARGUMENTS                  (* calling an account *)
| ContractDelegateCall of CALL_ARGUMENTS              (* library call *)
| ContractCreate of CREATE_ARGUMENTS          (* deploying a contract *)
```

[3] https://github.com/pirapira/eth-isabelle/tree/wtsc01

```
| ContractFail                              (* failing back to the caller *)
| ContractSuicide  (* destroying itself and returning to the caller *)
| ContractReturn of list byte                   (* returning to the caller *)
```

This definition describes everything that an Ethereum contract can do. When a high level language is designed for Ethereum, it's desirable that the language can cause all of these actions. Moreover, since the input-output interface is defined in an interactive theorem prover, the actions can be universally (∀) or existentially (∃) quantified in logical formulas that specify Ethereum smart contracts.

5 Formalizing the Deterministic Contract Execution

The Yellow Paper [26] specifies EVM's behavior uniquely for all possible inputs (either a contract creation or a message call) coming from external accounts. After no state transitions, the resulting state is left ambiguous. The original purpose of such determinism is to prevent the nodes from disagreeing, but the determinism also simplifies the formalization. We were able to formalize consecutive execution of instructions in our contract as a total function that produces a state. The deterministic definitions of the program semantics occupy 2,000 lines of Lem code. The determinism also made it straightforward to test this part of the EVM definition against a standard test suite (Subsection 5.3)[4].

We initially tried to implement EVM available in the latest Ethereum network. During the VM tests we found that EVM should price instructions differently depending on block numbers, so we modeled this as to pass the tests.

5.1 Defining Execution Contexts

During the formalization, we have identified the runtime state of EVM. While EVM is executing an account's code, EVM has access to the stack, the memory, the memory usage counter, the storage, the program counter, the balances of all accounts, the caller, the value sent along the current call, the data sent along the current call, the initial state kept for reverting into, the external account that originated the transaction, the codes on all addresses, the current block, the remaining gas, existence of accounts, and the list of touched storage indices. Everything except the last one is necessary for EVM execution. The last piece spares enumerating all storage indices while testing. These data are packed into a record type variable_ctx. Moreover, EVM can read the program and the address of the currently running contract. These data are packed into a record type constant_ctx.

An instruction can result in the following cases:

```
type instruction_result =
| InstructionContinue of variable_ctx (* the execution continues. *)
| InstructionAnnotationFailure              (* annotation was false. *)
```

[4] If nondeterminism existed in the EVM execution, at least, we would need to choose a representation of nondeterminism that works both in interactive theorem provers and in OCaml.

```
| InstructionToEnvironment of
   contract_action                                        (* the contract's move *)
  * storage                                          (* the new storage content *)
  * (address → w256)                    (* the new balance of all accounts *)
  * list w256          (* the list of possibly changed storage indices *)
  * maybe (variable_ctx * integer * integer)       (* continuation *)
```
The asterisk $*$ composes the type of *tuples*.

5.2 Defining Deterministic Contract Execution

Using the above definitions, we can define a function that operates an instruction on the execution environments:

```
val instruction_sem : variable_ctx → constant_ctx → inst →
instruction_result
```

```
let instruction_sem v c inst1 =
 subtract_gas (meterGas inst1 v c)
 (match inst1 with
 | Arith ADD → stack_2_1_op v c (fun a b → a + b)
 | Arith ADDMOD → stack_3_1_op v c
   (fun a b divisor →
    (if divisor = 0 then 0
     else word256FromInteger ((uint a + uint b) mod (uint divisor))))
 ⋮
 end)
```
where meterGas calculates the exact gas consumption of the executed instruction. We can repeat the semantics of single instructions to define the semantics of a whole program (`JUMP` instruction is not special because all instructions, including `JUMP`, change the program counter).

The type PROGRAM_RESULT is similar to INSTRUCTION_RESULT. The program semantics takes artificial step counters that disallow infinite execution because, in Isabelle/HOL, every function must be provably terminating[5]. This does not cause imprecision because any actual execution can be simulated with a sufficiently large step counter value.

```
val program_sem : variable_ctx → constant_ctx → int → nat →
program_result
```
During the modeling, we found that the Yellow Paper computes gas differently from the implementations. The subtlest case was the computation of gas for memory accesses: when a contract accesses the memory on addresses spanning from $2^{256} - 255$ to 1, the gas calculation differed in the Yellow Paper and in implementations. The Yellow Paper used 1 as the maximal touched address while all checked implementations used $2^{256} + 1$ instead. Since all implementations agreed, we filed a fix in the Yellow Paper.

[5] We can guarantee termination by the gas, but the proof is non-trivial (currently 980 lines of Isabelle code).

5.3 Testing the Deterministic Contract Interpreter

We tested our definition against a test suite called VM tests [2]. The test suite (together with other test suits) keep different Ethereum implementations conformant. We used VM tests to ensure conformance of our EVM definition. Lem automatically translated the definition into OCaml. The OCaml translation was then combined with a test case runner we wrote in OCaml (Fig. 3).

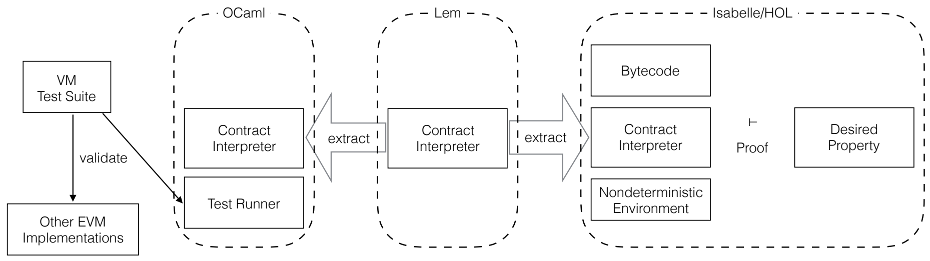

Fig. 3. Our Lem definition can be extracted into OCaml and Isabelle/HOL. We tested the OCaml extraction against the standard VM test suite. Using the Isabelle/HOL extraction, we proved safety properties about some bytecodes. In this figure, the VM test suite and other EVM implementations are not our contributions.

During the testing, we uncovered problems like:

- wrong word-to-integer conversion during `ADDMOD` in our EVM definition;
- different endianness between OCaml extraction and Isabelle/HOL extraction, due to our wrong direction; and
- small mistakes in the Yellow Paper, in most cases about modulo-2^{256}.

The number of successful test cases is 40,619 while no tests fail. We skipped 24 test cases because they involve running multiple contracts, and we chose to model only a single contract's execution deterministically. Running these 24 cases would involve major enhancements in our test runner: emulating multiple instances of our EVM model and communication among them.

In addition, we measured the code coverage of the VM test suite on the generated OCaml code. We found that `DELEGATECALL` instruction is never called, that `CALL` instruction is never called with insufficient balance to be transferred, that some instructions were never called with insufficient stack elements, and that the gas calculation after the latest changes is not tested. Although recent protocol changes are often tested in other test suites, the VM test suite can be complemented with these cases.

6 Formalizing the Nondeterministic Environment

We define the nondeterministic environment as a binary relation between a prestate of type (ACCOUNT_STATE * PROGRAM_RESULT) and a poststate of type

(ACCOUNT_STATE * VARIABLE_CTX). This binary relation encodes the environment's freedom. The binary relation is parametrized with an invariant (to be speculated by the verification practitioner) of the contract under verification, which limits the state changes on the contract during reentrancy. If this limitation makes the same speculated invariant provable, the invariant can be deemed established following an informal argument given in Subsection 6.3.

6.1 Implicit Balance Changes

We assume that the balances of accounts change freely while our contract is not executed. This assumption subsumes the payment for the gas. The storage of other accounts might change too. However, the balance of our contract is assumed not to decrease when there are no calls being executed on it[6].

On the other hand, the balance of our contract might increase when another account executes `SUICIDE` instruction, specifying our account as the recipient of the remaining balance. So the environment can freely increase the balance of our contract. We are assuming that the balance increase does not overflow (which seems to hold currently because the total balance of all accounts is below 2^{80} while the balances can be counted up to $2^{256} - 1$).

6.2 Gas Consumption During Calls

When our contract calls an account, the available amount of gas might decrease. We modeled this as a completely nondeterministic change. This treatment admits the actual gas decrease as one possibility, and it shortens the proof goals during brute-force proving. Without this treatment, during the symbolic execution described in Sec. 7, we saw the symbolic states grow rapidly because the remaining gas was represented as a long sequence of subtractions. With the nondeterministic choice, the remaining gas in the symbolic state is reduced into one variable after each call.

6.3 Modeling of Reentrancy as an Adversarial Environment's Step

We have freedom: the nested execution under reentrancy can either be a part of the system or the environment. The choice influences the proof structure. If the reentrancy is part of the system, proofs of safety properties need to explore all possibilities in the nested reentrant calls. If the reentrancy is part of the environment, the reentrancy is an adversarial step that changes the account state in some arbitrary ways. We chose the latter way because this matches better with the syntax of EVM bytecode, and it serves as the first approximation before building a bigger EVM definition involving call stacks.

We assume that the reentrancy can change the contract's account state (the balance and the storage) following a speculated invariant. Using this assumption,

[6] This property can be established only by checking all lines in the Yellow Paper that changes the balance.

we prove the same invariant on the outer call. If we finish proving this, we can perform mathematical induction over the number of nesting reentrancy to check that all message calls keep the invariant. This mathematical induction has not been formalized in any interactive theorem provers only because substantial development is required before stating the goal.

6.4 Cleanup of an Account after Self-Destruction

When a contract executes `SUICIDE` instruction, the storage and the code of the account are cleared not immediately but at the end of a transaction. The timing of this cleanup is determined by the adversarial environment. However, we know that the cleanup does not occur while a contract is still running.

7 Example Verification of Smart Contracts

To show the utility of our definitions, we have developed three example proofs in Isabelle/HOL.

Invariant of a Program that Always Fails As the shortest example, we prepared a smart contract that always fails. We proved that the code remains intact forever; in other words the contract does not execute `SUICIDE` operations.

Invariant of a Program that Fails on Reentrance The next example features reentrancy, which enabled an external account "to put ∼\$60M under her control" [5] during "the DAO" incident, where a coding mistake in a contract allowed leakage of the fund. We implemented a contract (Fig. 4) that calls an account but fails on reentrance. We proved that its storage values always stay within the specified values (Fig. 5) even when reentrant calls are attempted.

```
abbreviation fail_on_reentrance_program :: "inst list"
where
"fail_on_reentrance_program ==
  Stack (PUSH_N [0]) # Storage SLOAD # Dup 1 # Stack (PUSH_N [2]) #
  Pc JUMPI # Stack (PUSH_N [1]) # Arith ADD # Stack (PUSH_N [0]) #
  Storage SSTORE # Stack (PUSH_N [0]) # Stack (PUSH_N [0]) #
  Stack (PUSH_N [0]) # Stack (PUSH_N [0]) # Stack (PUSH_N [0]) #
  Stack (PUSH_N [0xabcdef]) # Stack (PUSH_N [30000]) # Misc CALL #
  Arith ISZERO # Stack (PUSH_N [2]) # Pc JUMPI # Stack (PUSH_N [0]) #
  Stack (PUSH_N [0]) # Storage SSTORE # Misc STOP # []"
```

Fig. 4. An Ethereum smart contract that calls an account but fails on reentrancy. The expression in this figure defines a list of instructions in Isabelle/HOL. See the Yellow Paper [26] for intuitive descriptions of instructions.

```
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)] ⟹
     account_killed st = False ⟹
     vctx_pc ve = 28 ⟹ vctx_storage ve 0 = 1 ⟹
     vctx_storage_at_call ve 0 = 0 ⟹
     fail_on_reentrance_invariant st"
```

Fig. 5. An invariant of the contract that fails on reentrancy, expressed in Isabelle/HOL. The whole invariant is a disjunction of two clauses: `depth_zero` holds when the contract is not running while `depth_one` holds when the contract has called an account.

Safety Property of a Compiled Program We proved a safety property of a realistic Ethereum contract with 501 instructions produced by the Solidity compiler. The safety property states that, if the storage has a flag set, only the owner recorded in the storage can decrease the balance or change the storage.

The proof is a brute-force symbolic execution in Isabelle/HOL. The proof contains repetitive 5,000 lines. It takes three hours for Isabelle to check the proof. There is huge room of improvements. Since the contract contains no loops, it should be possible to automate the whole proof. The proof checking time would be much shorter with more advanced techniques that appear in the next section.

8 Related Work

The idea and the techniques in this paper are not new, except that we apply these to EVM. Boyer and Yu [9] used a theorem prover Nqthm to model MC68020 processor, and checked correctness of a binary search implementation. Fox [10] modeled the ARM6 micro-architecture, which is far more complex than EVM, in HOL and validated it against the instruction set architecture. The deterministic part of our EVM definition happens to be in the form of functional big-step semantics [20] although our proof development is not advanced enough to enjoy its merits. The idea of combining theorem proving and testing is not new either even in the industry [7].

The literature suggests our future paths as well. Myreen, Fox and Gordon [18] defined Hoare logic for ARM machine code. Myreen, Gordon and Slint [17] further developed techniques for decompiling machine code with loops into recursive HOL functions. The approach of Kennedy et al. [11] is to formalize the machine code and then to build gradually structured programming method in Coq. Alternatively, we might try to build a higher level language that compiles into EVM. Jinja (Jinja is not Java) [12] demonstrates language specification and implementation in Isabelle/HOL. CakeML [13] is a programming language defined in Lem with a verified compiler into x86-64.

Some automatic analysis tools have been developed for Ethereum smart contracts. Oyente [15] implements abstract interpretation of EVM in Python with constraint solving using Z3. The tool can automatically detect several classes of vulnerabilities with false positives. Removing these classes of vulnerabilities does not guarantee lack of bugs. The tool does not implement all instructions. Bhargavan et al. [8] define translations from a fragment of Solidity and from EVM into F$*$, a functional programming language with a rich type system. They can detect diversion from certain programming disciplines in Solidity. They can also estimate an upper bound of gas consumption of an EVM program. They do not mention testing their translations against implementations[7].

9 Challenges and Future Work

Currently, verifying a realistic contract take around three hours on a Lenovo Ideapad 500S. Most of the time is spent in out-of-gas failures at various points in the program. One way to improve the situation is to set up a semantics that squashes all out-of-gas failures as a single case.

Another direction is to make the reasoning compositional. In other words, we should enable carrying over verification of small program snippets into verification of larger programs. This involves developing a syntax for properties (program logics) that is robustly concise during the compositional reasoning. Some program logics for machine code exist: e.g. Tan and Appel [24] and Myreen [18].

We have not tested the nondeterministic parts of our development. Also we have not validated our development against the blockchain history of the Ethereum network. The executable part of our model is considerably smaller than the whole EVM. If we model the whole EVM, we can try more standard test suites on our EVM definition. The modelling of the whole EVM would be the first step towards implementing a reference EVM out of our definitions.

The interactive theorem provers are designed for honest users. When a proof assistant admits a theorem that looks like falsehood, the proof assistant is called *Pollack-super-inconsistent*. Coq and Isabelle are known to be Pollack-super-inconsistent with auxiliary definitions and notations [25]. When falsehood seems provable, subtler errors can also creep in. For protecting users from malicious verification results, we need faithful presentation of the proven properties.

[7] One of the authors explained that the work had been done in a hackathon and the codebase had not been touched since.

For verifying smart contracts in more human-friendly languages, we can either formalize existing languages or build a compiler gradually in a theorem prover. The first approach poses the burden of developing and maintaining an up-to-date machine-readable definition of the language. The second approach poses the burden of integration with the ecosystem, where the contracts need to interface with JavaScript libraries and where developers need to be familiarized.

10 Conclusion

We defined EVM so that interactive theorem provers can reason about Ethereum smart contracts. Our EVM definition contains all instructions. We used our EVM definition in Isabelle/HOL and proved safety properties and invariants of Ethereum contracts in the presence of reentrancy. As a side effect, we discovered several problems in the specification; we requested eleven fixes to the Yellow Paper. We found thirteen code paths in our model that the VM test suite did not touch. We demonstrated formal executable specification is effective for verifying smart contracts, for testing the specification, and for measuring code coverage of virtual machine tests. We expect our development to be a basis for more sophisticated smart contract verification frameworks and for verified compilers from/to EVM bytecode.

Acknowledgments. We thank Sami Mäkelä for enabling Isabelle and Coq extractions and finding mistakes in our formalization. We thank Sidney Amani, Christian Reitwießner and the anonymous referees for their time and valuable comments on this paper.

References

1. The Coq proof assistant. https://coq.inria.fr/, accessed: 2016-12-19
2. Ethereum VM tests. https://github.com/ethereum/tests/tree/develop/VMTests, accessed: 2017-01-02
3. Solidity 0.4.8-develop documentation. https://solidity.readthedocs.io/, accessed: 2016-12-19
4. WebAssembly. http://webassembly.org/, accessed: 2016-12-16
5. Atzei, N., Bartoletti, M., Cimoli, T.: A survey of attacks on Ethereum smart contracts. Cryptology ePrint Archive (2016), http://eprint.iacr.org/2016/1007
6. Batty, M., Owens, S., Sarkar, S., Sewell, P., Weber, T.: Mathematizing C++ concurrency. SIGPLAN Not. 46(1), 55–66 (2011)
7. Becker, H., Crespo, J.M., Galowicz, J., Hensel, U., Hirai, Y., Kunz, C., Nakata, K., Sacchini, J.L., Tews, H., Tuerk, T.: Combining mechanized proofs and model-based testing in the formal analysis of a hypervisor. In: FM 2016. pp. 69–84. Springer (2016)
8. Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Gollamudi, A., Gonthier, G., Kobeissi, N., Kulatova, N., Rastogi, A., Sibut-Pinote, T., Swamy, N., Zanella-Béguelin, S.: Formal verification of smart contracts: Short paper. pp. 91–96. PLAS '16, ACM (2016)

9. Boyer, R.S., Yu, Y.: Automated proofs of object code for a widely used microprocessor. J. ACM 43(1), 166–192 (1996)
10. Fox, A.: Formal specification and verification of ARM6, pp. 25–40. Springer (2003)
11. Kennedy, A., Benton, N., Jensen, J.B., Dagand, P.E.: Coq: The world's best macro assembler? pp. 13–24. PPDP '13, ACM (2013)
12. Klein, G., Nipkow, T.: A machine-checked model for a Java-like language, virtual machine and compiler. ACM Trans. Progr. Lang. Syst. 28(4), 619–695 (2006)
13. Kumar, R., Myreen, M.O., Norrish, M., Owens, S.: CakeML: A verified implementation of ML. pp. 179–191. POPL '14, ACM, New York, NY, USA (2014)
14. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM 52(7), pp. 107–115 (2009)
15. Luu, L., Chu, D.H., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. pp. 254–269. CCS '16, ACM (2016)
16. Mulligan, D.P., Owens, S., Gray, K.E., Ridge, T., Sewell, P.: Lem: Reusable engineering of real-world semantics. SIGPLAN Not. 49(9), 175–188 (2014)
17. Myreen, M.O., Gordon, M.J.C., Slind, K.: Decompilation into logic–improved. In: FMCAD 2012. pp. 78–81 (2012)
18. Myreen, M.O., Fox, A.C.J., Gordon, M.J.C.: Hoare logic for ARM machine code, pp. 272–286. Springer (2007)
19. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: a proof assistant for higher-order logic, vol. 2283. Springer (2002)
20. Owens, S., Myreen, M.O., Kumar, R., Tan, Y.K.: Functional big-step semantics. In: ESOP'16. pp. 589–615. Springer (2016)
21. Ramananandro, T., Dos Reis, G., Leroy, X.: Formal verification of object layout for C++ multiple inheritance. SIGPLAN Not. 46(1), 67–80 (2011)
22. Roşu, G., Şerbănuţă, T.F.: An overview of the K semantic framework. Journal of Logic and Algebraic Programming 79(6), 397–434 (2010)
23. Slind, K., Norrish, M.: A brief overview of HOL4, pp. 28–32. Springer (2008)
24. Tan, G., Appel, A.W.: A compositional logic for control flow. In: VMCAI 2006. pp. 80–94. Springer (2006)
25. Wiedijk, F.: Pollack-inconsistency. Electron. Notes Theor. Comput. Sci. 285, 85–100 (2012)
26. Wood, G.: Ethereum: A secure decentralised generalised transaction ledger–EIP-150 revision. http://paper.gavwood.com/, accessed: 2016-12-19