CSF 2018, IEEE, pp 204-217. 2018

KEVM: A Complete Formal Semantics of the Ethereum Virtual Machine

Everett Hildenbrandt[ac]
hildenb2@illinois.edu

Manasvi Saxena[ac]
msaxena2@illinois.edu

Nishant Rodrigues[b]
nishant2@illinois.edu

Xiaoran Zhu[ae]
xiaoran@illinois.edu

Philip Daian[cd]
phil.daian@runtimeverification.com

Dwight Guth[c]
dwight.guth@runtimeverification.com

Brandon Moore[c]
brandon.moore@runtimeverification.com

Daejun Park[ac]
daejun.park@runtimeverification.com

Yi Zhang[ac]
yi.zhang@runtimeverification.com

Andrei Ştefănescu[c]
andrei@stefanescu.io

Grigore Roşu[ac]
grosu@illinois.edu

***Abstract*—A developing field of interest for the distributed systems and applied cryptography communities is that of smart contracts: self-executing financial instruments that synchronize their state, often through a blockchain. One such smart contract system that has seen widespread practical adoption is Ethereum, which has grown to a market capacity of 100 billion USD and clears an excess of 500,000 daily transactions.**

Unfortunately, the rise of these technologies has been marred by a series of costly bugs and exploits. Increasingly, the Ethereum community has turned to formal methods and rigorous program analysis tools. This trend holds great promise due to the relative simplicity of smart contracts and bounded-time deterministic execution inherent to the Ethereum Virtual Machine (EVM).

Here we present KEVM, an executable formal specification of the EVM's bytecode stack-based language built with the $\mathbb{K}$ Framework, designed to serve as a solid foundation for further formal analyses. We empirically evaluate the correctness and performance of KEVM using the official Ethereum test suite [1]. To demonstrate the usability, several extensions of the semantics are presented and two different-language implementations of the ERC20 Standard Token are verified against the ERC20 specification. These results are encouraging for the executable semantics approach to language prototyping and specification.

I. Introduction

The practical and academic success of Bitcoin [2], one of the early cryptocurrencies, has spawned a wide search for potentially promising applications of blockchain technologies. These blockchains and the blockchain-based systems they create tackle a wide range of disparate problems, including currency [2] [3], [4], distributed storage [5], academic research on consensus protocols [6], and more [7].

One such system, Ethereum, implements a general-purpose replicated "world computer" with a quasi-Turing complete programming language [8]. One goal of Ethereum is to allow for the development of arbitrary applications and scripts that execute in blockchain transactions, using the blockchain to synchronize their state globally in a manner that is fully verifiable by any system participant. Participants and contracts in the Ethereum system transact in a distributed currency known as Ether. Accounts on the Ethereum network can be associated with programs in a virtual-machine based language called the Ethereum Virtual Machine (EVM), described in the Yellow Paper, a semi-formal semantics and specification [9].

These programs are called "smart contracts", and execute when a transaction calls the account. Among other features, these contracts can tally user votes, communicate with other contracts, store or represent tokens and digital assets, and send or receive money in cryptocurrencies, without requiring trust in any third party to faithfully execute the contract [10] [11]. The computation and state are all public[1].

The growing popularity of smart contracts has led to increased scrutiny of their security. Bugs in such contracts can be financially devastating to the involved parties. An example of such a catastrophe is the DAO attack [12], where 150 million USD worth of Ether was stolen, prompting an unprecedented hard fork of the Ethereum blockchain [13]. Worse still, the DAO is one of many smart contracts which did not execute as expected, inducing costly losses [14], [15], [16], [17].

In fact, many classes of subtle bugs exist in smart contracts, ranging from transaction-ordering dependencies [18] to mishandled exceptions [18]. Further complicating the problem of obtaining high assurance, the EVM supports inter-contract execution to allow re-use of code via calls to library contracts.

To address these issues in a principled manner, we make use of the $\mathbb{K}$ Framework [19], [20]. $\mathbb{K}$'s goal is to separate construction of analysis tools from specification of particular programming languages, making it easier to construct correct tools based on a language's specification. From a $\mathbb{K}$ definition, many development tools are derived including a parser, interpreter, debugger, symbolic execution engine, and deductive verifier. We believe this paradigm for rapidly prototyping languages is particularly suitable to Ethereum; the EVM is continuously evolving and it is important that the specification

a University of Illinois at Urbana-Champaign: Computer Science
b University of Illinois at Urbana-Champaign: Mathematics
c Runtime Verification, Inc.
d Cornell Tech, IC3
e East China Normal University: Soft/Hardware Co-design Engineering Research Center

[1]Though all information is public, accounts on the network are pseudonyms.

Contract name	Value	Root cause
Parity Multisig 1 [16]	$200M	Private function exposure
Parity Multisig 2 [17]	$165M	Private function exposure
The DAO* [12]	$150M	Re-entrancy
SmartBillions [21]	$500K	Broken caching mechanism
HackerGold (HKG)* [22]	$400K	Typo in code

TABLE I: Smart contract failures impacting $\geq$ 400k USD. Stars* indicate implementations of the ERC20 API [23]. (Ether price data from https://coinmarketcap.com.)

of and tools around the EVM evolve along with it. Perhaps more importantly, the specification should be held to the same high testing standards as all official implementations of the EVM, disqualifying any non-executable formalisms.

A. Motivations

Smart contracts are ideal targets for verification, as they are small, terminating, deterministic programs. Moreover, these programs manage large amounts of Ether (often worth in excess of 100M USD), and exploits lead to Ether being transferred irreversibly [2]. Since all contract code is public, attackers can probe the system with full knowledge, testing and refining their attack privately before deploying it publicly.

All this means that there are huge financial incentives to attack the network, and all actors in the ecosystem are assumed adversarial – from the users submitting transactions, to the miners processing them, and even the nodes relaying them to the network. Indeed, Table I identifies and categorizes several contracts which have experienced high-profile exploits. Most of, if not all, these failures and others [14] could have been prevented through the use of formal analysis tools, saving the smart contract ecosystem hundreds of millions in past (and likely future) losses. Fortunately, it also means that developers are more inclined to put in the extra effort required to formally verify contracts. While there once was a dearth of usable software quality-assurance tools to safeguard against these attacks, the community is moving towards an ecosystem where formal analysis is not uncommon.

To tackle this complex mix of demand for high assurance and a rich adversarial model, the community has turned to formal methods, even issuing open calls for tool proposals [24] as part of what has been described as a "push for formal verification" [25]. In these proposals, the Ethereum Foundation has specifically called for "a human- and machine-readable formalization of the EVM, which can also be executed", "developing formally verified libraries in EVM bytecode or Solidity", and "developing a formally verified compiler for a tiny language" [24]. In this paper, we present the headway we have made into tackling these problems.

B. Challenges in Formalizing the EVM

Though the EVM is assembly like, it has several features which distinguish it from typical assembly languages and complicate the formalization. We briefly touch upon two of the challenges that are unique to the EVM.

- During the execution cycle of individual opcodes, the EVM has several points which can throw exceptions. Section 9.4.2 of the Yellow Paper [9] documents how exceptions can be thrown. There, it is specified "that no instruction can, through its execution, cause an exceptional halt" and an execution model which checks for exceptional states before executing is proposed. Initially, we attempted to follow this model, but soon realized that several additional points in the execution cycle can throw exceptions. To handle this, we re-built the semantics on top of a custom exception-based control-flow machine unique to the EVM, described fully in section III.
- In order to prevent Denial of Service (DoS) attacks caused by infinite computations, execution of EVM opcodes consumes a finite resource called "gas". As the EVM has evolved, this gas model has changed to facilitate and/or discourage the use of specific opcodes. To account for this, we took great care to make KEVM parametric in the particular chosen fee schedule. This means that KEVM actually defines a family of EVMs, one for each chosen fee schedule (see section III-E for more details).

C. Contributions

We present a formalization of the EVM in $\mathbb{K}$. This specification is:

- Unambiguous: $\mathbb{K}$ itself is based on sound foundations in Reachability Logic, leaving little room for dispute about the specification's meaning.
- Readable: $\mathbb{K}$ code itself is readable with some experience, and this project utilizes literate programming to make the specification as accessible as possible.
- Executable: A reference interpreter and debugger is generated from the specification.
- Faithful: Using the derived interpreter, we execute and pass the official test suites for EVM implementations.
- Performant: In section IV, we demonstrate the practicality of our automatically generated interpreter in a performance comparison with available specialized EVM tools.
- Formally Useful: The derived deductive verifier is demonstrated briefly on real world smart contracts (see section V). Our proofs to date are at https://github.com/runtimeverification/verified-smart-contracts.

All of these factors together have led to the Ethereum Foundation considering adopting our work as an official specification of the EVM (see section VI-A for details). To further KEVM's utility to developers and researchers, we provide a complete web-based semantics at https://jellopaper.org.

II. Background

We now provide some required background for our work, including a high-level overview of smart contracts, the EVM smart contract language, and the $\mathbb{K}$ Framework.

A. Ethereum

Ethereum [8], like Bitcoin, is a public blockchain transaction ledger. While Bitcoin's blockchain only stores transactions that exchange Bitcoin between addresses, Ethereum's blockchain holds addresses with EVM code. Transactions

```
contract Token {                                  JUMPDEST
 mapping(address=>uint) balances;                 PUSH F*40
 function deposit() payable {                     CALLER
  // (msg is a global representing                AND
  //  the current call)                           PUSH 0
  balances[msg.sender] += msg.value;              SWAP1
 }                                                DUP2
                                                  MSTORE
 function transfer(address recipient,             ...
  uint amount) returns(bool success) {            PUSH 40
   if (balances[msg.sender] >= amount) {          SWAP1
     balances[msg.sender] -= amount;              SHA3
     balances[recipient] += amount;               DUP1
     return true;                                 SLOAD
   }                                              CALLVALUE
   return false;                                  ADD
 }                                                SWAP1
}                                                 SSTORE
```

Fig. 1: Simple Solidity Token smart contract (left) and excerpt of compiled EVM deposit function (right).

recorded on the blockchain are invocations to the aforementioned code, and contain information about the data passed to the program as input. These programs are interpreted by a limited virtual machine called the Ethereum Virtual Machine (EVM) and are expressed in its corresponding language. This language is assembly-like, stack based, quasi-Turing complete and consists of 65 unique opcodes [9].

B. Smart Contracts

Smart contracts are computer programs which execute through blockchain transactions that are able to hold state, interact with decentralized cryptocurrencies, and take user input. The blockchain however, only stores EVM bytecode, which is too low level for development. Contracts are often written in a High Level Language like Solidity[26] or Viper[27] and compiled to EVM bytecode before blockchain deployment.

One example of a smart contract is shown on the left side of Figure 1. This contract represents a simplification of an on-chain token, a cryptographic asset able to be transferred and exchanged between users[2]. A mapping called `balances` stores an association between a user's address (derived from a private key that is required to authorize transactions from that account) and a balance in our example token. The user is able to deposit Ether into this contract with the `deposit` function, which is correspondingly marked payable. On deposit, the balances array for the sender of the transaction is increased by the amount of the deposit, minting new supply for our token. There is also a transfer function which allows users who have balance in the system to transfer tokens to other accounts, decreasing their balance and increasing the receiver's balance.

This is a simplistic contract with several flaws, including the presence of potentially unexpected arithmetic overflow in both functions and the lack of a withdraw function which means Ether is never withdrawable from the contract. Nevertheless, it illustrates the important features of the smart contract platform, including the ability to manipulate world-readable contract state (via the `balances` mapping) and process decentralized cryptocurrencies programmatically.

[2]On-chain tokens are custom currencies implemented as ledgers on the Ethereum network. Their value is not directly correlated to the value of Ethereum, but they use the Ethereum network for consensus.

C. Ethereum Virtual Machine (EVM)

Figure 1 also shows an annotated excerpt of our example Solidity token contract compiled to EVM. Specifically, this excerpt reads the balance of the sender from the contract's storage / world state, adds the value of the current call to this contract's balance (creating new tokens in exchange for Ether sent to the contract), and stores this new sum back into the relevant entry of the `balances` mapping. Addressing in the global storage for maps is based on the SHA3[3] hash of the map's offset in the contract and the key being looked up.

To prevent programs from executing indefinitely, the sender of each transaction pays a fee to the miners of their smart contract interaction on the blockchain (miners are users sequencing these transactions into blocks in the blockchain). This fee is charged proportional to how much gas is used by the contract. The fee schedule for execution is fully agreed upon by the network, and each transaction specifies a maximum amount of gas it is willing to use, as well as its exchange rate between Ether and gas. If a transaction runs out of gas during execution, it is aborted, its state updates are reverted, and miners keep the transaction's gas fees. This places an execution bound on all EVM transactions, enforcing termination, and allows the network to charge transactions proportional to the computational cost they incur.

D. The $\mathbb{K}$ Framework

The $\mathbb{K}$ Framework is a rewriting based framework for defining executable semantic specifications of programming languages, type systems and formal analysis tools. Given the syntax and semantics of a language, $\mathbb{K}$ generates a parser, an interpreter, as well as formal methods analysis tools such as a model checker and a deductive verifier. This avoids duplication while improving efficiency and consistency. For example, using the interpreter, one can test the semantics immediately, which significantly increases the efficiency of and confidence in semantics development. The verifier uses the same internal model for verifying programs, and that confidence carries over. Verification is discussed in section V and [28].

There exists a rich literature on using $\mathbb{K}$ for defining languages, including an online tutorial [29]. $\mathbb{K}$ has been used to formalize large languages like C [30] [31], Java [32] and JavaScript [33], among others. We will introduce $\mathbb{K}$ by need, as we discuss our formalization of the EVM in the next section.

III. $\mathbb{K}$ Semantics of EVM (KEVM)

A $\mathbb{K}$ specification includes three main components:

- a syntax of the language, supplied in an EBNF style,
- a description of the state/configuration, and
- the transition rules which drive execution of programs.

Here, we describe the KEVM definition structure as an introduction to both the EVM and to $\mathbb{K}$. Only the main semantics files `data.md` and `evm.md` are explained in depth.

[3]The actual hash function used is Keccak256, slightly different than SHA3.

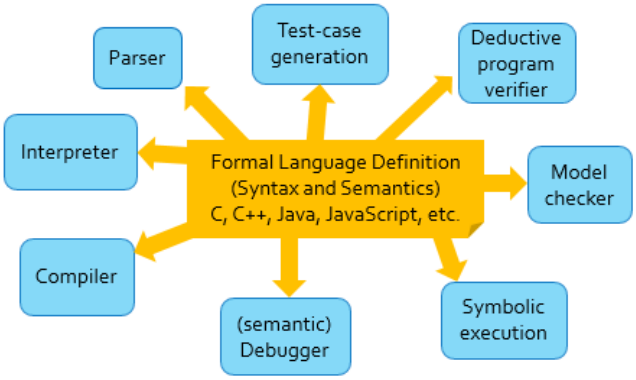

Fig. 2: The $\mathbb{K}$ approach as described in [28]

A. Utilities and Data Structures

Large parts of the KEVM definition are made simpler by defining simple data-structures and functions over them. Here several examples of this functional data are provided to introduce readers to $\mathbb{K}$ syntax.

Below we declare a new term `pow256` of builtin sort `Int`. The keyword `syntax` adds new productions to the grammar of terms. Since it is a "pure" function (that is, evaluation of `pow256` does not depend on surrounding context) we declare it with the `[function]` attribute.

```
    syntax Int ::= "pow256" [function]
// ----------------------------------
    rule pow256 => 2 ^Int 256
```

The semantics of `pow256` is defined using a rewrite rule (with keyword `rule`). If a term under consideration matches the pattern to the left of the rewrite arrow (`_=>_`), it is replaced with the pattern to the right. Here, `_^Int_` is a builtin $\mathbb{K}$ function that implements integer exponentiation. [4]

These patterns may be symbolic, i.e. contain variables. If $\mathbb{K}$ finds an assignment to the variables that makes the left hand side of the rule match a term, it replaces the matched term to the right hand side after applying the assignment. A `requires` clause can be supplied with a `rule` to state additional boolean pre-conditions to the rule firing. For example, `chop(_)`, defined below, is used throughout the EVM semantics to ensure that EVM words stay within the 256 bit width.

```
    syntax Int ::= chop ( Int ) [function]
// --------------------------------------
    rule chop ( I:Int ) => I %Int pow256
      requires I <Int 0 orBool I >=Int pow256

    rule chop ( I:Int ) => I
      requires I >=Int 0 andBool I <Int pow256
```

Here, the (seemingly redundant) `requires` clauses are an optimization to enhance performance (`%Int` is expensive) and simplify queries made to $\mathbb{K}$'s SMT solver (currently Z3 [34]) when doing symbolic reasoning. Once again, `_%Int_`, `_>=Int_` and `_<Int_` are builtin $\mathbb{K}$ operators for `Int`.

[4] To facilitate discussing operators used in the semantics, we implicitly give each operator a name with argument positions are replaced with underbars `_`. For example, the top-level operator in `3 +Int 4` is named `_+Int_`.

Because the EVM operates over 256 bit words, we implement functions for modulo arithmetic operations using `chop(_)`. Here, operators `_+Word_` and `_/Word_` are provided as examples. Note that EVM specifies that division by zero results in a zero value [9] (which, controversially, is meant to save gas).

```
    syntax Int ::= Int "+Word" Int [function]
                 | Int "/Word" Int [function]
// ----------------------------------------
    rule W0 +Word W1 => chop( W0 +Int W1 )
    rule W0 /Word 0  => 0
    rule W0 /Word W1 => chop( W0 /Int W1 )
      requires W1 =/=K 0
```

EVM is a stack-machine, meaning that wordstacks (stacks of words) are a necessary part of the definition. We implement a cons-list data structure for sort `WordStack` with empty element `.WordStack` and cons operator `_:_`. Also shown, the `WordStack` append operator `_++_` concatenates two stacks:

```
    syntax WordStack
      ::= ".WordStack" | Int ":" WordStack
        | WordStack "++" WordStack [function]
// ----------------------------------------
    rule .WordStack ++ WS' => WS'
    rule (W : WS)   ++ WS' => W : (WS ++ WS')
```

The definition of operator `_++_` here is entirely analogous to the definition of the cons-list append function from many functional languages. Indeed, the functional subset of $\mathbb{K}$ (when the keyword `function` is added to productions), is very similar to normal functional programming.

B. Representing State (Configuration)

$\mathbb{K}$ represents program execution state using a configuration. The configuration is an unordered list of (potentially nested) cells, specified in $\mathbb{K}$ using an XML-like notation.

When declaring transitions (as rewrites) over this state, any subset of the cells present in the configuration can be mentioned. This allows the user to specify only the necessary parts of the state for a given transition, letting $\mathbb{K}$ assume that the remaining parts of the configuration remain unchanged.

The KEVM configuration is split into two components: that of an active VM (for executing transactions and contracts), and the state of the network as a whole (e.g. account information). We omit the full configuration, which contains 70+ cells.

a) VM state: Execution of EVM programs must maintain the executing account (`<id>` cell), the current program counter (`<pc>` cell), and the current program (as a map from program counters to opcodes in the `<program>` cell). The `<wordStack>` and `<localMem>` cells provide memory in the form of a bounded wordstack and scratchpad RAM, respectively. The `<gas>` cell maintains how much longer execution can continue before the VM forcibly terminates the program (to avoid DoS attacks). The comments to the right of the cells indicate the names used in the Yellow Paper [9] for these components of the state.

```
configuration
 <k> $PGM:EthereumSimulation </k>
 <evm>
  <id>          0           </id>          // I_a
```

```
  <program>    .Map        </program>    // I_b
  <pc>         0           </pc>         // \mu_pc
  <wordStack>  .WordStack  </wordStack>  // \mu_s
  <localMem>   .Map        </localMem>   // \mu_m
  <gas>        0           </gas>        // \mu_g
  ...
</evm>
...
```

When declaring a configuration (using the `configuration` keyword), the initial values are supplied. $\mathbb{K}$ replaces the `$PGM` placeholder with the code for the program being run, usually specified as a command line parameter or input file. In the case of the test suite, this is a JSON object describing the network state at the time of execution, the code for the program being executed, and a set of post conditions expected at the end of execution. Traditionally, the `<k>` cell is used to drive execution and holds the next execution step of a $\mathbb{K}$ semantics.

b) Network state: The Ethereum blockchain forms a log of transactions on the network, which when replayed lead to the current world/network state. In our semantics, we choose to store the current world/network state over the append-only log of transactions leading to this state. In a given state, there may be any number of active accounts and pending transactions.

Here we show part of the `<network>` sub-configuration, specifically the portion corresponding to account states.

```
configuration
  ...
  <network>
    <activeAccounts> .Map </activeAccounts>
    <accounts>
      <account multiplicity="*" type="Map">
        <acctID>   0           </acctID>
        <balance>  0           </balance>
        <code>     .WordStack  </code>
        <storage>  .Map        </storage>
        <nonce>    0           </nonce>
      </account>
    </accounts>
    ...
  </network>
```

The `<accounts>` cell holds information about the accounts on the blockchain. Each `<account>` holds the accounts associated `<balance>`, `<code>` (smart contract), `<storage>` (persistent memory), and `<nonce>`[5]. By adding attribute `multiplicity="*"`, we state that 0 or more `<account>` cells can exist at a time (that is, multiple accounts can exist at a time on the network). As an optimization, we additionally state that accounts can be treated internally as a map from their `<acctID>` (by specifying `type="Map"` on the `<account>` cell and listing `<acctID>` as the first sub-cell)[6].

C. Execution

a) Exception-based control-flow: Exceptions are part of the low-level control-flow of the EVM – they may occur in case of invalid opcodes, `JUMP`s to PCs that haven't been marked as `JUMPDEST`s, if there is insufficient gas to pay for execution, or in case of stack over/under-flow. We built a simple imperative language for throwing/catching exceptions in KEVM. Exceptions consume anything following them on the `<k>` cell until they are caught.

```
    syntax KItem      ::= Exception
    syntax Exception ::= "#exception" | "#end"
// ---------------------------------------
    rule EX:Exception ~> (_:Int    => .)
    rule EX:Exception ~> (_:OpCode => .)
```

Note that here, the `<k>` cell is not explicitly mentioned in the rule; when no cell is mentioned and the operator being rewritten is not a `function`, it's assumed that the rule applies only at the front of the `<k>` cell. The operator `_~>_` ships with distributions of $\mathbb{K}$ and acts as an associative binary sequencing operation (read as "followed by", and similar to the semicolon in many imperative languages).

Here, we show how exceptions consume any following `Int` or `OpCode` by rewriting them to the empty computation (`.`) (which is the empty/identity element of the operator `_~>_`). These rules can be read as "when something of sort `Exception` is at the front of the `<k>` cell, it dissolves anything of sort `Int` or `OpCode` following". Note that the rewrite arrow (`_=>_`) scope here is *local*: matching happens on the entire rule but the state change only happens inside the parentheses.

To use exceptions for control flow, we provide a branching choice operator `#?_:_?#` which chooses the first branch when no exception is thrown and the second when one is.

```
    syntax KItem ::= "#?" K ":" K "?#"
// ----------------------------------
    rule                #? B1 : _  ?# => B1
    rule #exception ~> #? _  : B2 ?# => B2
```

The anonymous variable (`_`) is used to tell K that we do not care about the a subterm value when matching or rewriting.

b) Execution Cycle: Execution in KEVM is driven by the internal operator `#next`, which loads and triggers execution of the next opcode. As described in section 9.4 of the Yellow Paper [9], execution of a single opcode follows these steps:

1) Perform quick checks for exceptional opcodes.
2) Execute the opcode if the checks passed.
3) Increment the program counter.
4) Revert state in case of any exceptions.

Here is the $\mathbb{K}$ rule which gives semantics to the `#next` operator, performing the above steps:

```
rule <k> #next
      => #pushCallStack ~> #exceptional? [ OP ]
                        ~> #exec         [ OP ]
                        ~> #pc           [ OP ]
      ~> #? #dropCallStack : #popCallStack ?#
     ...
     </k>
     <pc> PCOUNT </pc>
     <program> ... PCOUNT |-> OP ... </program>
```

Note that this rule reaches across multiple cells in the configuration (including the `<k>`, `<pc>`, and `<program>` cells). The ellipsis (`...`, called *structural frames*) in these rules are not omission of details, but syntax supported by $\mathbb{K}$ for abstracting uninteresting parts of the state. We match on the

[5]This nonce is a globally accessible monotonically increasing integer value that counts the number of transactions performed by this account.

[6]This adds the extra requirement that any access of an `<account>` cell in a rule **must** mention the corresponding `<acctID>` cell.

current program counter PCOUNT along with the corresponding key/value pair anywhere in the program map to retrieve the next opcode OP. The operator _|->_ is $\mathbb{K}$'s builtin map-binding operator, which creates one key/value pair of the Map sort. Another rule in the semantics handles the case where the PCOUNT key is not present in the <program> map indicating that the program has run to termination.

Upon successfully finding the next opcode, first the current execution state is saved with internal operator #pushCallStack. Then, steps 1, 2, and 3 from above are performed using internal operators #exceptional?, #exec, and #pc. If an exception is thrown at any point during this, it consumes everything up to the choice operator #?_:_?# and takes the second branch which reverts the execution state with a call to #popCallStack. Given that no exception is thrown, the saved-off state is instead forgotten with a call to #dropCallStack (to save memory).

D. Example OpCodes

Opcodes are declared with sort corresponding to their arity to simplify the process of loading arguments from the <wordStack>. For example, BinStackOps consume two Words from the <wordStack>.

```
    syntax BinStackOp ::= "SUB" | "DIV"
// ------------------------------------
    rule SUB W0 W1 => W0 -Word W1 ~> #push
    rule DIV W0 W1 => W0 /Word W1 ~> #push
```

SUB and DIV perform simple arithmetic on their arguments then use internal operator #push to push the result onto the <wordStack>. In contrast, the opcodes SLOAD and SSTORE access the current accounts <storage>. In both cases, the current account ACCT is matched so that the appropriate <account> cell is selected for matching.

Here, SLOAD grabs a single word at position I from the <storage> and #pushes it onto the wordstack. A second rule (omitted here) specifies the behavior when the index I does not exist in the current account storage.

```
    syntax UnStackOp ::= "SLOAD"
// ----------------------------
    rule <k> SLOAD I => V ~> #push ... </k>
         <id> ACCT </id>
         <account>
           <acctID> ACCT </acctID>
           <storage> ... I |-> V ... </storage>
           ...
         </account>
```

SSTORE is used to write value V to index I in the current account <storage>. Notice that here, rewrite arrows are present in three cells: <k>, <storage>, and <refund>. This notational convenience allows users to specify rules more compactly, without having to duplicate parts of the configuration that remain unchanged on both the left and right-hand sides of the rule. The updates to state in each of the cells happens simultaneously, *only if* the overall left-hand side matches and the requires clause (if present) is met.

```
    syntax BinStackOp ::= "SSTORE"
// ------------------------------
    rule <k> SSTORE I V => . ... </k>
         <id> ACCT </id>
         <account>
           <acctID> ACCT </acctID>
           <storage>
           ... I |-> (OLD => V) ...
           </storage>
           ...
         </account>
         <refund> R =>
           #ifInt OLD =/=Int 0 andBool V ==Int 0
            #then R +Word Rsstoreclear < S >
            #else R
           #fi
         </refund>
         <schedule> S </schedule>
```

Here, a quirk of the EVM is also demonstrated with the <refund> cell. If the OLD value in <storage> is non-zero but the new value V is zero, the current executing account is refunded some gas for freeing up memory. Notice that the refund amount, Rsstoreclear, is parametric over S (the current fee <schedule>) as explained further in section III-E.

E. Gas Semantics

Each execution step and memory expansion in EVM costs some state-dependent amount of gas, which ensures that all computations are terminating. KEVM mimics the Yellow Paper's gas calculation by providing several gas helper functions defined in Appendix G in [9].

For example, the function Csstore calculates the gas needed to store a value to an account's storage:

```
  syntax Int
    ::= Csstore (Schedule, Int, Int) [function]
//-------------------------------------------
  rule Csstore(SCHED, V, OLD)
    => #ifInt V =/=Int 0 andBool OLD ==Int 0
         #then Gsstoreset   < SCHED >
         #else Gsstorereset < SCHED >
       #fi
```

Note that the cost of storing to an account's memory depends on whether you are setting it for the first time (before it was zero, now it's not) or not. Beyond that, each gas-cost is parametric over a *fee schedule*. As Ethereum has evolved the fees for each opcode have been tweaked to disincentivize behavior expensive to the network and incentivize alternatives. This means that the same computation may consume different amounts of gas depending on when (in which block of the blockchain) it is executed. Since the blockchain requires that all past transactions be replayable, all EVM implementations must be aware of all previous schedules and not just the one currently in use. We abstract this information into schedules. Above, Gsstoreset and Gsstorereset are parametric over a Schedule; the function _<_> allows making a schedule constant parametric over a schedule.

```
  syntax Int
  ::= ScheduleConst "<" Schedule ">" [function]
//---------------------------------------------
```

Here we show some example fee schedules and schedule constants:

```
   syntax ScheduleConst
     ::= "Gzero"    | "Gbase"   | ...
```

```
      | "Gbalance" | "Gsload" | ...
// ---------------------------------

   syntax Schedule ::= "DEFAULT"
// -----------------------------
   rule Gzero < DEFAULT > => 0
   rule Gbase < DEFAULT > => 2

   syntax Schedule ::= "EIP150"
// ----------------------------
   rule Gbalance < EIP150 > => 400
   rule Gsload   < EIP150 > => 200
```

The fee schedule to use is set through the command line flag `-cSCHEDULE=<FEE_SCHEDULE>`. This allows us to execute and verify programs against any appropriate fee schedule.

IV. Quantitative Semantics Evaluation

A. Correctness and Performance

As consensus-critical software, implementations of EVM are held to a high standard; past disagreements have caused accidental forks of the blockchain which leads to disparate world-views [35]. We based our semantics on the Yellow Paper [9], but found inconsistencies confirmed by its developers.

Test Set (no. tests)	**Lem EVM**	**KEVM**	**cpp-ethereum**
Lem (40665)	288:39	34:23	3:06
VMStress (18)	-	72:31	2:25
VMNormal (40665)	-	27:10	2:17
VMAll (40683)	-	99:41	4:42
GSNormal(22705)	-	35:00	1:30
GSQuad (250)	-	855:24	0:21
GSAll (22955)	-	889:00	1:51

TABLE II: Lem EVM vs KEVM vs cpp-ethereum

Table II shows a performance comparison between KEVM, the Lem semantics [36], and the C++ reference implementation distributed by the Ethereum foundation[7]. The Lem semantics (discussed more in section VII) is the only other executable formal specification of the EVM we are aware of.

All execution times are given as the full sequential CPU time (in MM:SS format) on an Intel i5-3330 processor (3GHz on 4 hardware threads) and 24 GB of RAM. By comparing to the C++ reference implementation, we show the feasibility of using the KEVM formal semantics for prototyping, development, and test-suite evaluation.

The row Lem indicates a run of all the tests that the Lem semantics can run (a subset of the VMTests). The row VMStress indicates a run of all 18 stress tests in the test-suite, to compare the performance of KEVM with the C++ implementation. The row VMNormal is a run of all the non-stress tests in the test-suite (*not* the same set of tests as Lem). VMAll is the addition of the second and third rows and is included for completeness. The last three rows indicate a runs of the GeneralStateTests; GSNormal are the non-stress tests, GSQuad are the stress tests, and GSAll is the addition of the two. Under the GeneralStateTests, our tools performs well except in the case of QuadraticStateTests (250 out of 22955).

[7]https://github.com/ethereum/cpp-ethereum

As shown in the comparison, the automatically extracted interpreter for KEVM outperforms the currently available formal executable EVM semantics. KEVM compares favorably to the C++ implementation, performing under 30 times slower on the stress tests, roughly 20 times slower on all tests, and only 11 times slower on the Lem and VMNormal tests.

B. Implementation Effort

The time to develop the first release-quality KEVM was roughly 2 months of light activity by 2 developers followed by 3 months of heavy activity by 4 developers.

The overall definition is broken into three major modules:

- File `data.md` provides module `EVM-DATA`, which defines defines EVM data-structures (742 lolc[8]).
- File `evm.md` provides module `EVM`, which defines the EVM state, opcodes, and execution cycle (2645 lolc).
- File `driver.md` provides the extra module `ETHEREUM-SIMULATION`, which is largely used for running the test suites (744 lolc).

The total count of non-blank and non-literate lines of code for KEVM comes in at 2644. For comparison, the reference C++ implementation weighs in at 4588 lines of code. This measurement was taken on commit-hash ee0c6776c of https://github.com/ethereum/cpp-ethereum by counting non-blank lines of all `*.h` and `*.cpp` files in subdirectory `libevm`.

We argue that these numbers are not atypical for implementing an interpreter for a small real-world programming language, not to mention the extra tools that $\mathbb{K}$ provides for analysis and security along the way.

V. Verification of Smart Contracts

A primary motivation for this work has been to mitigate security failures as listed in section I-A. Many such issues can be addressed via verification, i.e. proving a program conforms to a formal property. As mentioned in section II-D, $\mathbb{K}$ generates a deductive verifier. We briefly describe the verifier, and demonstrate verification of real-world contracts. For a complete list of the proofs we have to date, see [37].

A. $\mathbb{K}$'s Deductive Verifier

As a prelude to explaining smart contract verification using our semantics, we need briefly cover the deductive verifier's theoretical foundations. At its core, $\mathbb{K}$'s deductive verifier performs automated Reachability Logic (RL) reasoning [28]. RL is a sound and relatively complete logic tailored for reasoning about reachability. Program correctness specifications in RL are expressed as *reachability claims*. A reachability claim is a sentence of the form $\phi \Rightarrow \psi$ (read ϕ *reaches* ψ), where ϕ are ψ are formulae in Matching Logic (ML) [38]. We briefly describe ML, and ML's formulae (known as *patterns*) using an example. Let's consider the rule for `chop`, from section III-A

```
rule chop ( I:Int ) => I
  requires I >=Int 0 andBool I <Int pow256
```

This rule can be written as $\alpha \wedge \beta \Rightarrow \alpha'$, where α, β, α' are patterns `chop(I:Int)`, `I <Int 0 andBool I >=Int`

[8]lolc = lines of literate code

`pow256` and `I` respectively. The `requires` construct in $\mathbb{K}$ introduces the pattern β as a side condition, resulting in the pattern $\alpha \wedge \beta$. This pattern can be viewed as the set of configurations that match α structurally and satisfy β logically.

The reachability claim $\phi \Rightarrow \psi$ specifies that the set of states represented by pattern ϕ will either reach a state in ψ or not terminate when executed with the given language semantics. $\mathbb{K}$'s deductive verifier treats both operational semantics rules and program correctness specifications as reachability rules. It then uses RL's proof system (figure 3) to derive the proof claims using the semantic rules as axioms. Thus, when instantiated with the semantics in section III, $\mathbb{K}$'s verifier provides a sound procedure for reasoning about EVM programs. We direct the reader to [28] for details on Reachability Logic, and the proof search algorithm used by the $\mathbb{K}$ verifier.

a) Expressive Power: A Hoare triple $\{Pre\}Code\{Post\}$ can be represented as the reachability claim $\widehat{Code} \wedge \widehat{Pre} \Rightarrow \epsilon \wedge \widehat{Post}$, where ϵ is a pattern representing the empty program. $\widehat{Code}$ is a minimal state pattern containing the *Code* but with program variables replaced with logical variables. Similarly for $\widehat{Pre}$ and $\widehat{Post}$ variables are replaced with logical counterparts.

Using this, we can directly encode functional correctness and safety properties as reachability claims for the prover to discharge. Often times individual correctness properties are not enough though; developers have high-level security goals regarding contract logic, perhaps even across multiple transactions. $\mathbb{K}$ can still be of use for these goals; many security faults are due to collections of correctness bugs (as opposed to poor design) which can be specified as multiple claims to be discharged simultaneously. Additionally, many trace properties can be captured by breaking a claim $\phi \Rightarrow \psi$ into several claims $\phi \Rightarrow \phi_1$, $\phi_1 \Rightarrow \phi_2$, ..., $\phi_n \Rightarrow \psi$, allowing proofs over intermediate states as well. Finally, though transaction execution is completely deterministic on the EVM, transaction ordering is a source of non-determinism which can be exploited by miners on the network. $\mathbb{K}$ reasons with *all*-path reachibility [28], meaning that reachability claims over multiple transactions can be proven with respect any ordering of the transactions.

B. ERC20 Token Standard

The ERC20 standard [23] is one of the most important standards for the implementation of tokens within Ethereum smart contracts. ERC20 provides basic functionality to transfer tokens and to be approved so they can be spent by another on-chain third party. ERC20-compliant tokens were responsible for raising and holding over one billion USD in the six months before the writing of this report. In the so-called "ICO rush", a series of ERC20-compliant token/coin launches were used to raise funding on the Ethereum platform [39].

ERC20 compliant tokens are therefore an attractive target for software verification. In order to evaluate the viability and usefulness of our semantics based verification approach, we decided to target implementations of the ERC20 token standard. Compliance with the ERC20 standard is also important if a contract wishes to be recognized by external software including wallets, exchanges, and other contracts expecting to interact with tokens. We focused on verifying the functional correctness of the following five functions from [23]:

- `balanceOf(address): uint`
  Retrieve the balance of the specified account.
- `transfer(address, uint): bool`
  Transfer tokens to the specified account.
- `transferFrom(address, address, uint): bool`
  Request a payment between two accounts.
- `approve(address, uint): bool`
  Approve payment requests up to the specified amount.
- `allowance(address, address): uint`
  Retrieve remaining allowed payment requests.

The implementation details of these methods are left to the user, with minimal semantic behavior provided in the specification, leaving room for a wide range of complex tokens (and the associated security vulnerabilities). Complete details of the ERC-20's interface is documented at [23].

Ideally, we would like to create a specification describing generally desirable properties of ERC20, usable in verifying a wide range of implementations at the EVM level. While these contracts are usually implemented in high-level languages, performing this verification at the EVM level removes the need for compiler trust. Note that compiler problems are real; one compiler bug required the re-issuance of a token holding and processing 190 million USD[9].

1) Challenges in Verifying EVM Programs: EVM lacks high level constructs including functions and explicit data types. Any invocation of an EVM program always begins with the VMs program counter set to 0. Higher level languages such as Solidity and Viper, however, have functions and types, allowing users to directly call functions in contracts without dealing with low-level EVM code. This lack of high level constructs makes writing specifications for programs at the EVM level tedious and error prone. For instance, writing a specification to reason about any of the ERC20 methods at the EVM level requires specifying the actual program counter values corresponding to the start and the end of the function.

We address this issue by using a DSL modeled on the Ethereum ABI. The Ethereum ABI [40] is a mechanism for simulating a function call at the EVM level. Contracts are written in higher level languages and call functions in other contracts. The ABI facilitates this by specifying the encoding and decoding rules for a transaction's calldata: an input field available to each transaction (realized in KEVM with the cell `<callData>`). To be ABI complaint, EVM bytecode must include logic at the beginning for handling control transfer (via `JUMP`s) to the program counter of the called function.

It is worth noting that the ABI is not a part of the protocol itself, and hence does not appear in the Yellow Paper [9]. However, compilers for common higher level languages like Solidity and Viper produce ABI compliant code.

[9]The vulnerability of the Serpent compiler and the associated reissuance of the Augur is described in https://medium.com/@AugurProject/serpent-compiler-vulnerability-rep-solidity-migration-5d91e4ae90dd.

Reflexivity :

$$\mathcal{A} \vdash \varphi \Rightarrow \varphi$$

Transitivity :

$$\frac{\mathcal{A} \vdash_{\mathcal{C}} \varphi_1 \Rightarrow^{+} \varphi_2 \qquad \mathcal{A} \cup \mathcal{C} \vdash \varphi_2 \Rightarrow \varphi_3}{\mathcal{A} \vdash_{\mathcal{C}} \varphi_1 \Rightarrow \varphi_3}$$

Axiom :

$$\frac{\varphi \Rightarrow \varphi' \in \mathcal{A}}{\mathcal{A} \vdash_{\mathcal{C}} \varphi \Rightarrow \varphi'}$$

Case Analysis :

$$\frac{\mathcal{A} \vdash_{\mathcal{C}} \varphi_1 \Rightarrow \varphi \qquad \mathcal{A} \vdash_{\mathcal{C}} \varphi_2 \Rightarrow \varphi}{\mathcal{A} \vdash_{\mathcal{C}} \varphi_1 \vee \varphi_2 \Rightarrow \varphi}$$

Abstraction :

$$\frac{\mathcal{A} \vdash_{\mathcal{C}} \varphi \Rightarrow \varphi' \qquad X \cap \mathit{FreeVars}(\varphi') = \emptyset}{\mathcal{A} \vdash_{\mathcal{C}} \exists X \ \varphi \Rightarrow \varphi'}$$

Circularity :

$$\frac{\mathcal{A} \vdash_{\mathcal{C} \cup \{\varphi \Rightarrow \varphi'\}} \varphi \Rightarrow \varphi'}{\mathcal{A} \vdash_{\mathcal{C}} \varphi \Rightarrow \varphi'}$$

Logic Framing :

$$\frac{\mathcal{A} \vdash_{\mathcal{C}} \varphi \Rightarrow \varphi' \qquad \psi \text{ is a (patternless) FOL formula}}{\mathcal{A} \vdash_{\mathcal{C}} \varphi \wedge \psi \Rightarrow \varphi' \wedge \psi}$$

Consequence :

$$\frac{\models \varphi_1 \rightarrow \varphi_1' \qquad \mathcal{A} \vdash_{\mathcal{C}} \varphi_1' \Rightarrow \varphi_2' \qquad \models \varphi_2' \rightarrow \varphi_2}{\mathcal{A} \vdash_{\mathcal{C}} \varphi_1 \Rightarrow \varphi_2}$$

Fig. 3: Sound and relatively complete proof system of Reachability Logic. $\mathcal{A}$ is the initial (trusted) execution semantics of the programming language (axioms). The $\mathcal{C}$ on $\vdash_{\mathcal{C}}$ indicates that the *circularities* $\mathcal{C}$ are reachability claims conjectured but not yet proved. The **Circularity** proof rule allows us to conjecture any to-be-proven reachability claim as a circularity, while **Transitivity** allows us to use the circularities as axioms (only after we have made progress on proving them).

The ABI specifies that the first four bytes of call data must be the `Keccak` hash (referred to as the function selector) of a canonical representation of the invoked function's signature. The canonical representation is a string consisting of function name and comma-separated list of types. From the fifth byte onward, the parameters are encoded based on their type. We briefly describe our DSL for the encoding.

```
syntax TypedArg ::= #uint160 ( Int )
                  | #uint256 ( Int )
// Others omitted for readability
syntax WordStack
::= #abiCallData( String , TypedArgs )
                                  [function]
```

The DSL's `#abiCallData` construct takes as input a function name, and a comma separated list of typed arguments. The function then generates the function selector bytes from the signature, and uses the type information from arguments to encode the argument data. For ABI-compliant EVM bytecode, we can use the ABI based DSL to bypass the process of specifying concrete program counter values and calldata corresponding to the function boundaries in our claims, which removes a major limitation of performing proofs at the EVM level. In section V-B2 and V-C, we demonstrate the usage of the ABI-DSL.

2) Procedure: The procedure to verify contracts using KEVM is largely the same each time:

i Write down the high-level logic of a contract as abstract state-updates on the configuration.
ii Compile the code to EVM for use with KEVM.
iii Fill in the configuration from step 1 with the code and corresponding calls.
iv Inspect the queries being made to the SMT solver to understand why the claims are not provable.
v Fix bugs discovered in the original code or in the claims to make the proof go further.
vi Goto step (iv) if the proof is not discharged yet.

3) Solidity Implementation: The HKG Token (an implementation of ERC20) was initially a topic of discussion when a vulnerability based in a typographical error lead to a re-issuance of the entire token [22], disrupting a nontrivial economy based on it. Previously, the token was audited by humans and deemed secure[10], further reinforcing the error-prone nature of the human review process and the need for tools.

Specifically, the typographical error in the HKG Token came in the form of an `=+` statement being used in place of the desired `+=` when updating a receiver's balance during a transfer. While typographically similar, these statements are semantically very different, with the former being equivalent to a simple `=` (the plus saying that the expression following should be treated as positive) and the latter desugaring to add the right hand quantity to the existing value in the variable on the left hand side of the expression. In testing, this error was missed, as the first balance updated would always work (with `balance =+ value` being semantically equivalent to `balance += value` when `balance` is 0, in both cases assigning `value` to `balance`). Even with full decision or branch coverage in testing, multiple transfers on the same account can be entirely omitted in a way that is difficult to notice through human review.

Even after fixing the bug described above though, we were unable to prove the functional correctness of the HKG contract. It turns out that only after specifying bounds on the inputs and gas supplied in the pre-condition that there would not be any arithmetic overflow or out-of-gas exceptions were we able to verify the contract. In addition, the HKG Token threw an exception if the `transfer` function was called with a 0 amount, which amounted to another pre-condition added to the specification. Note that this last issue is *non-compliance* with the ERC20 standard; the process of verifying functional correctness of the contract caught this error and the mentioned potential bugs.

The original proofs we performed were directly over the EVM, which was quite cumbersome and made it difficult to

[10]https://zeppelin.solutions/security-audits

find the exact start and end program-counter values to use. However, since these tokens were ABI-compliant, we were able to use our ABI DSL to later simplify the proof claims.

4) Viper Implementation: To test whether our ABI-level specification of the ERC20 was modular across implementations from different high-level languages, we also verified a Viper implementation of the ERC20 released by the Ethereum Foundation. The token we verified is available at [37].

Three changes to the specifications were necessary for the proofs to go through:

- The gas needed to be changed. This is expected because the Viper and Solidity compilers generate different EVM.
- The Viper implementation correctly handles the 0 case for `transfer`, so we had to remove the added pre-condition.
- The HKG and Viper implementations have different behavior for `transfer` and `transferFrom`. If not enough funds are present, the HKG token returns false while the Viper token throws an exception. Both behaviors are allowable according to the ERC20 specification.

With these minor changes, the same ABI-level verification went through on the Viper token, demonstrating the modularity of our ABI-level verification approach.

C. Example Proof Claims

Due to space, we cannot include the full reachability claims here. The $\mathbb{K}$ prover accepts reachability claims in the same format as semantic rules. Instead of interpreting the supplied module as axioms (like the modules in the semantics itself), the module is interpreted as a set of reachability claims.

Here we summarize one of the claims, including interesting parts of the state (but omitting uninteresting bits with ...). In the following claim, any symbol starting with a `%` indicates a constant which has been replaced by a symbol for clarity. In particular, `%HKG_Program` is the EVM bytecode for the HKG token program, `VALUE` represents the symbolic amount to transfer, `B1` and `B2` are the starting balances of accounts 1 and 2, respectively, and `A1` is the allowance of account 1[11].

Here, we also make use of the ABI DSL to populate the `<callData>` cell with the intended contents. Without this DSL, we would have needed to find the starting and ending program counters for function `transfer` manually. In addition, we would have needed to manually encode the correct function to call and its arguments.

```
rule
 <k> #execute => (RETURN _ _ ~> _) </k>
 ...
 <program>  %HKG_Program </program>
 <pc>       0 => _       </pc>
 ...
 <callData>
  #abiCallData ( "transfer"
               , #address(%CALLER_ID)
               , #uint256(VALUE)
               )
 </callData>
 ...
```

[11] `pow256` is defined in section III-A.

```
<accounts>
  <account>
   <acctID>    %ACCT_ID      </acctID>
   <balance>   BAL           </balance>
   <code>      %HKG_Program  </code>
   <acctMap>  "nonce" |-> 0  </acctMap>
   <storage>
   ...
   (%ACCT_1_BALANCE |-> (B1 => B1 -Int VALUE))
   (%ACCT_1_ALLOWED |-> A1)
   (%ACCT_2_BALANCE |-> (B2 => B2 +Int VALUE))
   (%ACCT_2_ALLOWED |-> _)
   ...
   </storage>
  </account>
</accounts>

requires VALUE >Int 0
 andBool B1 >=Int 0 andBool B1 <Int pow256
 andBool B2 >=Int 0 andBool B2 <Int pow256
 andBool B2 +Int VALUE <Int pow256
 andBool B1 -Int VALUE >=Int 0
 andBool          VALUE <Int pow256
```

The claim above specifies that in all valid executions starting in the left-hand-side of the rule, either execution will never terminate or it will reach an instance of the right-hand-side. Specifically, this means that any transfer of amount `VALUE` from account 1 to account 2 (with `VALUE` sufficiently low and various overflow conditions met) will happen as intended in the execution of the `transfer` code provided. While one can rely on the EVM to throw exceptions on stack overflow or out-of-gas, arithmetic overflow will usually go undetected as the VM does not throw an exception on arithmetic overflow.

VI. Other Applications

As alluded to in Section II-D, besides the verification engine, $\mathbb{K}$'s semantics-first approach allows deriving several other tools. We now describe two such artifacts.

A. Jello Paper

While developing these semantics, a common problem we faced was interpreting the Yellow Paper, the English language specification of the EVM [9]. Often times, the Yellow Paper is unclear or underspecified, and in some exceptional cases even unfaithful to what actual implementations do.

For example, as mentioned in section I-B, Section 9.4.2 of [9] describes exceptions as if they are all detectable prior to opcode execution. While it may be possible to implement EVM in this way, it is not clear that this is the simplest or best way (as it would lead to duplicating computation). No implementations seem to work this way, casting further doubt on this description; instead exceptions are thrown when they happen. Our original implementation tried to do it in this way, eventually necessitating a redesign.

In other cases, the expected behavior is underspecified. For example, it is not always explicit about what should happen when an opcode attempts to access a non-existent account's data (as `EXTCODESIZE` and `EXTCODECOPY` may do). Another example is the appearance of "junk bytes" in a program's bytecode (which do not correspond to any opcodes); these

can be used for loading long immutable strings of data into the VM. Though not originally addressed in the Yellow Paper, the community has reached agreement on these issues.

In some places the Yellow Paper is even inaccurate. The DELEGATECALL instruction, with semantics given in Appendix H of [9], describes the gas provided to the caller as equal to $\mu_s[0]$ (the top of the <wordStack>). This is clearly incorrect, since $\mu_s[0]$ is a user-provided value, and the user could set it equal to $2^{256} - 1$, leading to the user having near infinite gas. The test suites and other implementations indicate that the intended behavior is to use C_{callgas} (as CALLCODE and CALL do), but with the $value$-transferred argument set to 0. For the same opcode, it describes the exceptional condition of not enough balance in terms of I_v, but in fact no value transfer occurs so this condition should never occur.

In the process of building an executable specification, all of these issues naturally arose when testing against the test-suite, as they did for other implementations. These problems make implementing tools and infrastructure for the Ethereum ecosystem needlessly error-prone and inefficient. Instead, we propose using our "developer" documentation, which is automatically generated from the KEVM semantics[12]. This version of the semantics, called the Jello Paper, is available at https://jellopaper.org. We hope to continue improving the Jello Paper readability, and have been in communication with members of the Ethereum Foundation regarding establishing it as a reference specification for the EVM platform and an executable successor to [9].

B. Gas analysis tool

EVM programs are forced to always terminate to prevent malicious actors from mounting a DoS attack on the network. This is done by allotting gas for execution ahead of time and charging each VM operation some gas. If gas is exhausted before execution finishes, an exception is thrown and the state is reverted. For many contracts, functional correctness is dependent upon enough gas being supplied up front. To help users decide how much gas they should supply, we extended the semantics with a gas analysis tool.

The semantics was already designed with extensibility in mind; execution is parametric over an extra <mode> cell which controls how to interpret EVM programs[13]. For example, in VMTESTS mode, execution of CALL and CREATE opcodes is not performed (as specified by the Ethereum Test Suite [1]).

```
configuration ...
              <mode> $MODE:Mode </mode>
              ...

syntax Mode ::= "NORMAL" | "VMTESTS" | ...
```

The file analysis.md adds the execution mode GASANALYZE, along with some modifications to the definition of the #next operator. In GASANALYZE mode, the #next operator executes normally until it hits a control-flow operator (eg. JUMP, JUMPI, or JUMPDEST), collecting the overall gas consumed to do so. At the control-flow operator, the overall gas consumed is recorded in the <analysis> cell (along with the starting and ending program counter for that basic block). Finally, the program counter is forcibly incremented past the control-flow operator, and the analysis is restarted.

In this way, each basic block is executed in isolation and the amount of gas used is collected and reported back to the user in the <analysis> cell. Note that the current implementation only calculates an approximation, but some engineering effort would result in a more accurate calculation.

This extension is a 1.8% increase in the size of the semantics (87 lines of literate code, roughly a day of work), demonstrating the flexibility of having a directly extendable executable specification of the EVM.

[12]The tool Sphinx (http://sphinx-doc.org) is used to generate the Jello Paper.

[13]The execution mode is set on the command line with -cMODE=<EXECMODE>.

Tool	Spec.	Exec.	Tests	Prover	Bugs	Gas
Yellow Paper	✓	✗	✗	✗	✗	✗
cpp-ethereum	✗	✓	✓	✗	✗	✗
Lem spec	✓	✓	✓	✓	✗	✗
Oyente	✗	✓	✗	✗	✓	✓
hevm	✗	✓	✗	✗	✗	✗
Manticore	✗	✓	✗	✗	✓	✓
REMIX	✗	✓	✗	✗	✓	✓
Dr. Y's	✗	✓	✗	✓	✗	✗
F*	✗	✓	✗	✓	✓	✗
KEVM	✓	✓	✓	✓	✗	✓

TABLE III: Feature comparison of EVM semantics and other software quality tool efforts.

VII. Comparison with Related Work

There has been substantial practical interest in formally verifying properties of smart contracts for the reasons we enumerate in Section II. For example, the Solidity IDE incorporates Why3 [41] (a semi-automated theorem prover) to help verify smart contracts written in the higher-level Solidity language. In this section, we compare the practical artifacts derived from and generated by our work to existing efforts. A list compiled by Dr. Yoichi Hirai[14] informs our comparison. We do not include or compare with any tools which operate over other languages (e.g., Solidity source analysis tools) exclusively, serve as implementations of an Ethereum network client, or are closed (eg. Securify [15]).

a) Feature Comparison Overview: The tools produced in the Ethereum community are meant to fill a variety of purposes, many of which are also able to be accomplished directly from our executable semantics. We choose the following metrics of comparison for the tools we list:

- **Spec.:** Suitable as a formal specification?
- **Exec.:** Executable on concrete tests?
- **Tests:** Passes the Ethereum test-suites?
- **Prover:** Serves as theorem prover for EVM?
- **Bugs:** Heuristic-based tools for finding bugs?
- **Gas:** Analyzes gas complexity of EVM programs?

Table III shows an overview of the results of our comparison. We briefly describe each effort and compare it to

[14]https://github.com/pirapira/awesome-ethereum-virtual-machine

[15]http://securify.ch

the relevant KEVM artifact. The projects fit two categories: semantic specifications and smart contract analysis tools.

A. Semantic Specifications

a) Yellow Paper: [9] The official document describing the execution of the EVM, as well as other data, algorithms, and parameters required to build consensus-compatible EVM clients and Ethereum implementations. It cannot be tested against the conformance test-suite; instead it serves as a guide for implementations to follow. Much of the machine definition is supplied as several mutually recursive functions from machine-state to machine-state. The Yellow Paper is occasionally unclear or incomplete about the exact operational behavior of the EVM; in these cases it is often easier to simply consult one of the executable implementations.

b) cpp-ethereum: [16] A C++ implementation that also serves as a de-facto semantics of the EVM. The Yellow Paper and the C++ implementation were developed by the same group early in the project, so the Yellow Paper conforms mostly to the C++ implementation. In addition, the conformance test-suite is generated from the C++ implementation. This means that if the Yellow Paper and the C++ implementation disagree, the C++ implementation is favored.

c) Lem semantics: [36] A Lem ([42]) implementation of EVM provides an executable semantics of EVM for doing formal verification of smart contracts. Lem compiles to various interactive theorem provers, including Coq, Isabelle/HOL, and HOL4. The Lem semantics does not capture intercontract execution precisely as it models function calls as non-deterministic events with an external (speculated) relation dictating the "allowed non-determinism". This semantics is executable and passes all of the VMTests test-suite except for those dealing with more complicated intercontract execution, providing high levels of confidence in its correctness.

d) GMS small-step specification: [43] A small-step specification of the EVM inspired by the EtherLite semantics of [18]. The specification is non-executable, but provides a precise guide for implementers of the EVM.

B. Smart Contract Analysis Tools

a) Oyente: [17] An EVM symbolic execution engine written in Python supporting most of the EVM. Many heuristics-based drivers of the engine are provided for bugfinding.

b) hevm: [18] A Haskell implementation of EVM including on an interactive debugger mode, which allows stepping through contract execution one opcode at a time.

c) Manticore: [44] A symbolic execution engine for virtual machines, including models for x86, x86_64, ARMv7, and EVM. This tool exports a Python API for specifying programs, driving symbolic execution, and checking assertions.

d) REMIX: [19] A JavaScript implementation of the EVM with a browser-based IDE for building and debugging smart contracts. Some static analysis is built into the tool, allowing it to catch pre-specified classes of bugs in smart contracts.

e) Dr.Y's Ethereum Contract Analyzer: [20] A symbolic execution engine for EVM to summarize the semantics of smart contracts. A debug mode allows step-by-step execution.

f) F⋆ formalization of EVM: [45] An implementation of the EVM in the F⋆ language[21] which passes roughly half of the VMTests at the time of writing. The same paper discusses an on-paper small-step specification of the EVM as well [43].

VIII. Future Work

We believe this rich ecosystem of tools, all generated programmatically from a single independent reference semantics, has the opportunity to be transformative in the development and deployment of secure smart contracts while avoiding a large class of potential losses and failures. With our existing semantics and EVM interpreter, we plan on finishing the work required to encourage the widespread adoption of our work as the reference semantics and interpreter for the EVM system.

a) Future EVM Hardforks: As demonstrated in section III-E, the current semantics is parametric over the selected fee schedule. The EVM will continue to evolve as new opcodes are added and gas prices are changed, and the specification will need to evolve with it. KEVM provides a solid tool for prototyping and testing updates to the EVM specification, allowing for a smoother protocol update process.

b) Analysis Tools: Another key direction is the formalization of classically known contract antipatterns, specifically those that have lead to exploits in the past. These antipatterns can be used to generate dynamic checkers for EVM programs, similar to the dynamic checker for C in [46], [47].

For example, to check for integer overflows in the execution of a program, the semantics can be modified to halt every time an integer number overflows by modifying the semantics of just the `chop` function (section III-A). At execution time, if the function `chop` is called, the user will be alerted that their smart contract may have an overflow. Other classic antipatterns (eg. stack over/underflow demonstrated in section V-B2) can be caught with similar small modifications to the semantics.

c) ABI-Level DSL: Currently our ABI-level DSL only supports statically-sized ABI types, but the ABI contains several dynamically sized types as well[22]. We plan to extend our DSL to support dynamic types too, allowing writing specifications over any ABI-compliant contract. Once the full ABI is supported, we will have produced an executable specification of the EVM ABI abstraction.

d) Verified EVM Libraries: To further assist others in building high-assurance contracts, we intend to provide fully verified library contracts. Over time, we'll collect a body of high-assurance EVM code for the community.

[16] https://github.com/ethereum/cpp-ethereum
[17] https://github.com/melonproject/oyente
[18] https://github.com/dapphub/hevm
[19] https://github.com/ethereum/remix
[20] https://blog.slock.it/an-ethereum-contract-analyzer-93e9da92fecb
[21] https://www.fstar-lang.org/
[22] The ERC20 contract uses only statically sized types.

IX. Conclusion

In this paper, we presented a formalization of the EVM using the $\mathbb{K}$ Framework. This provides a specification of the EVM, a reference interpreter, and a suite of tools for program analysis and verification. KEVM is the first executable specification of the EVM that completely passes official test-suites.

Not only is our semantics complete and faithful, but performant. This is an important point if the semantics is intended to be used in a CI (Continuous Integration) environment where every change should be tested thoroughly. Changes to the EVM can realistically be prototyped on KEVM, yielding both an updated specification and implementation.

Beyond being a specification of the EVM, KEVM serves as a platform for building a wide range of analysis tools and other semantic extensions for EVM. We demonstrated this with three extensions: a full web-based version of our semantics, a gas analysis tool, and ABI-level DSL for verification of real world contracts. Each of the extensions leveraged the existing semantics, meaning each required minimal code changes.

Finally, we verified solidity and viper implementations of the ERC 20 token standard against the official specification. This gave us a chance to test the robustness of our ABI-level DSL; indeed proving the Viper sources after the specification was developed for Solidity was much simpler. The specification provided can be used nearly as-is for verifying other proper implementations of ERC20s, serving as the basis for developing a library of verified EVM code.

Already, KEVM has proved to be a useful tool for software developers working on smart contracts. Beyond that, KEVM is a valuable resource to Ethereum developers experimenting with evolving the EVM and protocol updates, as it streamlines the process of bringing an implementation in line with the specification for experimentation purposes. We hope this serves to signal the practicality of an *executable specification first* approach to programming language design, as well as the merits of separating the construction of programming language semantics from analysis tools. We believe the application of these tools can drastically increase the rigor and security of both currently deployed and future smart contracts.

A. Acknowledgements

Many members of the Ethereum community have provided overwhelming support for this project, allowing us to quickly identify issues of key importance to the community at large. A special thanks goes to IOHK[23] for recognizing the importance of this effort and supporting it through generous funding and by connecting us to research communities around the world working on similar topics. Beyond that, we would like to thank the Initiative for CryptoCurrencies and Contracts (IC3)[24] for allowing us to present this work to the Ethereum community at large, which provided invaluable feedback on future directions to take this project.

Numerous colleagues also helped in developing these semantics, both in the initial design and idea phases and the later $\mathbb{K}$ programming phases. First we would like to thank Zane Ma (UIUC) and Deepak Kumar (UIUC) for initiating this project and bringing to our attention the need for formal analysis tools in the Ethereum community. At the 2017 IC3 Bootcamp, Lorenz Breidenbach (Cornell Tech, IC3, ETH Zürich) provided numerous suggestions about future directions to take this work and useful tools to build on the KEVM semantics. Yoichi Hirai and Vitalik Buterin of the Ethereum Foundation provided valuable feedback on a late version of this document, ensuring its coherence.

Finally, this work would not have been possible without the extensive help of the $\mathbb{K}$ teams, both the Formal Systems Lab at UIUC[25] and at Runtime Verification, Inc[26]. In particular, Cosmin Radoi and Xiaohong Chen assisted in using the $\mathbb{K}$ verification tools (and quickly fixed bugs we stumbled upon in $\mathbb{K}$).

The work presented in this paper was supported in part by the Boeing grant on "Formal Analysis Tools for Cyber Security" 2016-2017, the NSF grants CCF-1318191, CCF-1421575, CNS-1330599, CNS-1514163, and IIP-1660186, the NSF Graduate Fellowship under Grant No. DGE-1650441, an IOHK gift, and a grant from the Ethereum Foundation.

[23] https://iohk.io/
[24] http://www.initc3.org/
[25] http://fsl.cs.illinois.edu
[26] https://runtimeverification.com/

REFERENCES

[1] The Ethereum Foundation, "Ethereum tests," https://github.com/ethereum/tests, 2015.

[2] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008, https://bitcoin.org/bitcoin.pdf.

[3] A. Narayanan, J. Bonneau, E. Felten, A. Miller, and S. Goldfeder, Bitcoin and Cryptocurrency Technologies: A Comprehensive Introduction. Princeton University Press, 2016.

[4] E. B. Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza, "Zerocash: Decentralized anonymous payments from bitcoin," in Security and Privacy (SP), 2014 IEEE Symposium on. IEEE, 2014, pp. 459–474.

[5] S. Wilkinson, T. Boshevski, J. Brandoff, and V. Buterin, "Storj a peer-to-peer cloud storage network," 2014, https://storj.io/storj.pdf.

[6] E. K. Kogias, P. Jovanovic, N. Gailly, I. Khoffi, L. Gasser, and B. Ford, "Enhancing bitcoin security and performance with strong consistency via collective signing," in 25th USENIX Security Symposium (USENIX Security 16). Austin, TX: USENIX Association, 2016, pp. 279–296. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/kogias

[7] B. Ong, T. M. Lee, G. Li, and D. Chuen, "Evaluating the potential of alternative cryptocurrencies," Handbook of digital currency. Amsterdam: Elsevier, pp. 81–135, 2015.

[8] V. Buterin and Ethereum Foundation, "Ethereum White Paper," 2013, https://github.com/ethereum/wiki/wiki/White-Paper.

[9] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," 2014, (Updated for EIP-150 in 2017) http://yellowpaper.io/.

[10] N. Szabo, "Smart contracts," Unpublished manuscript, 1994, http://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart_contracts_idea.html.

[11] G. W. Peters and E. Panayi, "Understanding modern banking ledgers through blockchain technologies: Future of transaction processing and smart contracts on the internet of money," in Banking Beyond Banks and Money. Springer, 2016, pp. 239–278.

[12] P. Daian, "DAO attack," 2016, http://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/.

[13] M. del Castillo, "Ethereum executes blockchain hard fork to return DAO funds," 2016, http://goo.gl/MpwrhS.

[14] V. Buterin, "Thinking about smart contract security," 2016, https://blog.ethereum.org/2016/06/19/thinking-smart-contract-security/.

[15] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on Ethereum smart contracts." IACR Cryptology ePrint Archive, vol. 2016, p. 1007, 2016.

[16] L. Breidenbach, P. Daian, A. Juels, and E. G. Sirer, "An in-depth look at the parity multisig bug," 2017, http://hackingdistributed.com/2017/07/22/deep-dive-parity-bug/.

[17] J. Steiner, "Security is a process: A postmortem on the parity multi-sig library self-destruct," 2017, http://goo.gl/LBh1vR.

[18] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," Cryptology ePrint Archive, Report 2016/633, 2016, http://eprint.iacr.org/2016/633.

[19] G. Roşu and T. F. Şerbănuţă, "An overview of the K semantic framework," Journal of Logic and Algebraic Programming, vol. 79, no. 6, pp. 397–434, 2010, http://kframework.org/.

[20] Formal Systems Lab, UIUC, "The K framework," 2006, http://kframework.org.

[21] J. Solana, "$500K hack challenge backfires on blockchain lottery SmartBillions," 2017, https://calvinayre.com/2017/10/13/bitcoin/500k-hack-challenge-backfires-blockchain-lottery-smartbillions/.

[22] J. Manning, "Ether.camps hkg token has a bug and needs to be reissued," 2017, https://www.ethnews.com/ethercamps-hkg-token-has-a-bug-and-needs-to-be-reissued.

[23] The Ethereum Foundation, "ERC20 token standard," https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20-token-standard.md, 2017.

[24] C. Reitwiessner, "Dev update: Formal methods," 2016, https://blog.ethereum.org/2016/09/01/formal-methods-roadmap/.

[25] P. Rizzo, "In formal verification push, Ethereum seeks smart contract certainty," 2016, http://www.coindesk.com/ethereum-formal-verification-smart-contracts/.

[26] Ethereum, "Ethereum solidity documentation," 2017, https://solidity.readthedocs.io/en/develop/.

[27] ——, "Viper - new experimental programming language," 2017, https://github.com/ethereum/viper.

[28] A. Ştefănescu, D. Park, S. Yuwen, Y. Li, and G. Roşu, "Semantics-based program verifiers for all languages," in Proceedings of the 31th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'16). ACM, Nov 2016.

[29] FSL, UIUC, "K tutorial," http://www.kframework.org/index.php/K_Tutorial, 2012.

[30] C. Ellison and G. Rosu, "An executable formal semantics of c with applications," in Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'12). ACM, January 2012, pp. 533–544.

[31] C. Hathhorn, C. Ellison, and G. Roşu, "Defining the undefinedness of c," in Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15). ACM, June 2015, pp. 336–345.

[32] D. Bogdănaş and G. Roşu, "K-Java: A Complete Semantics of Java," in Proceedings of the 42nd Symposium on Principles of Programming Languages (POPL'15). ACM, January 2015, pp. 445–456.

[33] D. Park, A. Ştefănescu, and G. Roşu, "KJS: A complete formal semantics of JavaScript," in Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15). ACM, June 2015, pp. 346–356.

[34] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in International conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer Berlin Heidelberg, 2008, pp. 337–340.

[35] A. Quenston, "Ethereum's blockchain accidentally splits," 2016, https://www.cryptocoinsnews.com/ethereums-blockchain-accidentally-splits/.

[36] Y. Hirai, "Defining the ethereum virtual machine for interactive theorem provers," WSTC17, International Conference on Financial Cryptography and Data Security, 2017.

[37] KEVM and Runtime Verification, "Github: Verified Smart Contracts," https://github.com/runtimeverification/verified-smart-contracts, 2018.

[38] G. Roşu, "Matching logic," Logical Methods in Computer Science, 2017.

[39] S. Melendez, "Inside the ico bubble: Why initial coin offerings have raised more than $1 billion since January," 2017, http://goo.gl/7b5nCd.

[40] Ethereum, "Ethereum application binary support," 2017, https://github.com/ethereum/wiki/wiki/Ethereum-Contract-ABI.

[41] J.-C. Filliâtre and A. Paskevich, Why3 — Where Programs Meet Provers. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 125–128. [Online]. Available: https://doi.org/10.1007/978-3-642-37036-6_8

[42] D. P. Mulligan, S. Owens, K. E. Gray, T. Ridge, and P. Sewell, "Lem: reusable engineering of real-world semantics," in ACM SIGPLAN Notices, vol. 49, no. 9. ACM, 2014, pp. 175–188.

[43] I. Grishchenko, M. Maffei, and C. Schneidewind, "A semantic framework for the security analysis of ethereum smart contracts. technical report." 2018, https://secpriv.tuwien.ac.at/tools/ethsemantics.

[44] T. of Bits, "Manticore: Symbolic execution for humans," https://github.com/trailofbits/manticore, 2017.

[45] I. Grishchenko, M. Maffei, and C. Schneidewind, "A semantic framework for the security analysis of ethereum smart contracts," in Principles of Security and Trust, L. Bauer and R. Küsters, Eds. Cham: Springer International Publishing, 2018, pp. 243–269.

[46] P. Daian, D. Guth, C. Hathhorn, Y. Li, E. Pek, M. Saxena, T. F. Serbanuta, and G. Rosu, "Runtime verification at work: A tutorial," in Runtime Verification - 16th International Conference, RV 2016 Madrid, Spain, September 23-30, 2016, Proceedings, ser. Lecture Notes in Computer Science, vol. 10012. Springer, September 2016, pp. 46–67.

[47] D. Guth, C. Hathhorn, M. Saxena, and G. Rosu, "RV-Match: Practical semantics-based program analysis," in Computer Aided Verification - 28th International Conference (CAV 2016), Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I, ser. LNCS, vol. 9779. Springer, July 2016, pp. 447–453.