KEVM: A Complete Semantics of the Ethereum Virtual Machine

Everett Hildenbrandt (UIUC), Manasvi Saxena (UIUC), Xiaoran Zhu (ECNU and UIUC),
Nishant Rodrigues (UIUC), Philip Daian (Cornell Tech, IC3, and Runtime Verification Inc.),
Dwight Guth (Runtime Verification Inc.), and Grigore Roşu (UIUC and Runtime Verification Inc.)

August 17, 2017

Abstract

A developing field of interest for the distributed systems and applied cryptography community 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 secure approximately 30 billion USD of currency value and in excess of 300,000 daily transactions.

Unfortunately, the rise of these technologies has been marred by a repeated series of security vulnerabilities and high profile contract failures. To address these failures, the Ethereum community has turned to formal verification and program analysis which show great promise due to the computational simplicity and bounded-time execution inherent to smart contracts. Despite this, no fully formal, rigorous, comprehensive, and executable semantics of the EVM (Ethereum Virtual Machine) currently exists, leaving a lack of rigor on which to base such tools.

In this work, we present KEVM, the first fully executable formal semantics of the EVM, the bytecode language in which smart contracts are executed. We create this semantics in a framework for executable semantics, the $\mathbb{K}$ framework. We show that our semantics not only passes the official 40,683-test stress test suite for EVM implementations, but also reveals ambiguities and potential sources of error in the existing on-paper formalization of EVM semantics [45] on which our work is based. These properties make KEVM an ideal formal reference implementation against which other implementations can be evaluated.

We proceed to argue for a semantics-first formal verification approach for EVM contracts, and demonstrate its practicality by using KEVM to verify practically important properties over the arithmetic operation of an example smart contract and the correct operation of a token transfer function in a second contract. We show that our approach is feasible and not computationally restrictive. We hope that our work serves as the base for the development of a wide range of useful formally derived tools for Ethereum, including model checkers, certified compilers, and program equivalence checkers.

1 Introduction

The practical and academic success of Bitcoin [31], one of the early cryptocurrencies, has spawned a wide search for potentially promising applications of blockchain technologies. These blockchains and blockchain-based systems tackle a wide range of disparate problems, including currency [31] [32] [42], distributed storage [44], academic research on consensus protocols [23], and more [33].

One such system, Ethereum, implements a general-purpose replicated "world computer" with a quasi-Turing complete programming language [7]. 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 can be verified by any participant in the system. 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 EVM, which is described semi-formally in [45].

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 (or potentially external programs), 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 [43] [35]. The computation and state are all public[1], stored on the Ethereum blockchain.

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 recent DAO attack [11], where 50 million USD worth of Ether was stolen, prompting an unprecedented hard fork of the Ethereum blockchain [13]. Worse still, the DAO is only one of a number of smart contracts which did not execute as expected, leading to security violations and unexpected losses with immediate financial impact [6] [2].

In fact, many such classes of subtle bugs exist in smart contracts, ranging from transaction-ordering dependencies to mishandled exceptions [25]. Further complicating the problem of obtaining rigorous assurance for these contracts, the EVM supports inter-contract execution, allowing for rich network dynamics as contracts offload part of their computation to other contracts.

Tackling this complex mix between requirements for high assurance and a rich adversarial model, the Ethereum community has turned to formal verification, issuing open calls for formal verification proposals [37] as part of what has been described as a "push for formal verification" [38]. 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" [37]. We address the first two directly and lay the groundwork for the third in our work.

To address these goals on a principled manner, one attractive formal analysis framework is the $\mathbb{K}$ framework [41] (`http://kframework.org/`). The goal of $\mathbb{K}$ is to separate specification of analysis tools from specification of particular programming languages or other models, making it easier to specify both the analysis tools and the programming languages. We believe this paradigm is particularly suitable to Ethereum, as by definition EVM implementations must come to consensus on a single execution of a given transaction to drive the state transitions of the overall network [45]. With a diverse set of independent implementations that must agree on state (enumerated partially in Section 7), the need for a clean, formal, and unambigous definition of the EVM is clear. $\mathbb{K}$ allows us to provide such a formal definition that can be mechanically transformed to a reference interpreter for EVM and a range of formally rigorous analysis tools.

The primary contributions of this paper are as follows:

1. **EVM Semantics in $\mathbb{K}$:** We define a formally rigorous, executable instance of the EVM semantics in the $\mathbb{K}$ framework, covering all of the EVM instructions. Based on the formal semantics provided in [45], our semantics is able to find ambiguities present in the original seemingly formal but incomplete EVM specification, validating our approach. We release KEVM as open source, for use as a reference semantics in Ethereum client development.

2. **EVM Reference Interpreter:** Using our formal semantics, we automatically generate an EVM reference interpreter capable of executing EVM transactions and updating the EVM world state accordingly. We abstract away some details of the blockchain system itself, and provide a formally derived interpreter capable of running the full official Ethereum VM test suite with reasonable execution time. To our knowledge, this is the first full formally rigorous Ethereum interpreter that does not rely on a built-in hardcoded implementation of the EVM, and *the first formally specified EVM interpreter to pass the full 40,683-test EVM compliance test-suite.* Additionally, the derived interpreter demonstrates favorable performance, making it useful as a real-world testing platform.

3. **EVM Program Verifier:** We use the above to create a program verifier, and demonstrate the full verification of an example EVM program against a specification. The property specified includes both the functional correctness of the EVM program as well as the gas complexity of the program. In doing so, we demonstrate EVM software analysis tools rigorously derived from and backed by a formal, complete, and human-readable EVM semantics. We also show an example verification of a transfer function for a widely used token standard in the Ethereum ecosystem, ERC20.

[1]Though all information is public, accounts on the network can be anonymous.

```
contract Token {
  mapping(address=>uint) balances;
  function deposit() payable {
    // (msg is a global representing
    //  the current transaction)
    balances[msg.sender] += msg.value;
  }

  function transfer(address recipient,
   uint amount) returns(bool success) {
    if (balances[msg.sender] >= amount) {
      balances[msg.sender] -= amount;
      balances[recipient] += amount;
      return true;
    }
    return false;
  }
}
```

```
tag 8          //  function deposit() payable { ...
  JUMPDEST     // mark as function start (jumpable)
  PUSH F*40    // push mask for 20-byte address
  CALLER       // push msg.sender onto stack
  AND          // extract address type from mask+caller
  PUSH 0       // push 0 to stack
  SWAP1        // switch caller to top of stack
  DUP2         // duplicate value 0 to reuse
  MSTORE       // put caller in mem 0 for store lookup
  ...          // next lines push [0x0, 0x40] to top
  ...          // of stack (arguments for SHA3 lookup)
  PUSH 40      // 0x40 (64) mem bytes to read in SHA3
  SWAP1        // place mem address to SHA3 (0x0) first
  SHA3         // relevant storage addr (SHA3 of mem 0)
  DUP1         // duplicate address for read-then-write
  SLOAD        // read balances[msg.sender] from address
  CALLVALUE    // push msg.value onto stack
  ADD          // add two values above
  SWAP1        // swap sum with store addr (from SHA3)
  SSTORE       // store new computed sum
```

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

4. **Unified Toolset:** Many software analysis tools exist for the Ethereum ecosystem. We show how many such software analysis tools, which each encode their own models and semantics of the EVM, can be automatically generated from our single formal reference semantics. Generating these tools automatically from a single testable and executable semantics reduces the probability of tool errors caused by divergent models. We demonstrate how such a toolset may be implemented by building a simple gas analysis engine.

All of this work is open-source and provided at `https://github.com/kframework/evm-semantics`, and it is our hope that the Ethereum community will adopt our approach towards more rigorous, secure contracts.

2 Background

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

Ethereum [7] is a public, distributed ledger based on blockchain technology first proposed and popularized by Bitcoin. Whereas Bitcoin's blockchain only stores transactions that exchange Bitcoin between addresses, Ethereum's blockchain stores the complete execution state history of distributed programs. These programs are interpreted by a limited virtual machine known as the Ethereum Virtual Machine (EVM) and are expressed in its corresponding stack-based language. Like other stack-based VM languages, EVM binary consists of a series of statements as opcodes and their arguments, and is executed sequentially. Smart contracts, which are often written in higher level languages such as Solidity [17] or Serpent [16], are then compiled to EVM bytecode, a stack based, Turing-complete language, which consists of 65 unique opcodes [45]. Smart contracts consist of a *contract address*, contract *balance*, and program execution code and state.

2.1 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.

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 public 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, which decreases their balance and increases the corresponding receiver's balance.

While 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), it illustrates the important features of the smart contract platform, including the ability to manipulate world-readable and universally consistent contract state (the `balances` mapping) and process decentralized cryptocurrencies programatically.

2.2 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 transaction being sent to the contract to this 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 hash of the map's offset in the contract and the key being looked up, with full details on the operation of storage available at `https://github.com/androlo/solidity-workshop/tree/master/tutorials`.

To prevent programs from executing infinitely and to make sure all client applications of the smart contract system are fairly charged, the sender of each transaction must pay 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 its execution, it is aborted, all state updates are reverted, and the miners keep the full gas fees associated with the transaction. 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.

2.3 Security Problems

The need for security in smart contracts was identified early on [14]. Smart contracts provide the perfect storm of security problems (and thus the perfect testing grounds for formal analysis tools!) for a number of reasons:

- They are designed to store cryptocurrency, which when stolen can be transferred irreversibly [31], can be difficult to trace, and can be laundered effectively [29].
- The quantity of the money stored in these contracts tends to be high, with contracts often storing in excess of 100M USD[3], a strong attack incentive.
- As specified in [45], all contract code is stored publicly on the blockchain, allowing attackers to probe the system with full knowledge and test a range of attacks.
- The Ethereum environment is adversarial, with all actors from the miners involved in processing transactions to nodes involved in relaying assumed to be potentially malicious.

[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.

[3]Based on Ethereum-derived tokens on `http://coinmarketcap.com/`.

Contract name	Value affected	Root cause
The DAO* [11]	150M USD	Re-entrancy
HackerGold (HKG)* [26]	400K USD	Typo in code
Rubixi [6]	< 20K USD	Wrong constructor name
Governmental [6]	10K USD	Exceeds gas limit
Parity Multisig [5]	200M USD	Unintended function exposure

Table 1: Selected smart contract failures and their characteristics (Ether price data from coinmarketcap.com)

- The evolving Ethereum ecosystem suffers from a lack of usable software quality tools.

Table 1 shows several contracts which have experienced high-profile failures, identifying their root causes. Note that all of the mentioned failures are preventable with verification-based analysis tools. Contracts marked with a star comply with the ERC20 standard (see `https://github.com/ethereum/EIPs/issues/20` for information about the ERC20 standard), which specifies an API that "token" contracts must implement to integrate with exchange libraries and user interfaces.

As this figure shows, formal verification could have saved the smart contract ecosystem hundreds of millions in past (and likely future) losses. In addition to these losses, many of which were sourced from the analysis in [6], the analysis mentions "various instances" of lost funds because of fallback functions in other contracts exhausting a gas limit, and "various instances" of call stack limit exceptions. All are preventable through formal verification, and in this work we will demonstrate techniques to verify conformance of an EVM contract to an example specification. We further symbolically execute an ERC20-compliant transfer function through KEVM, which along with the capabilities of our prover establishes the foundation for a full EVM-level formalization of ERC20.

Smart contracts provide a uniquely attractive proving ground for formal techniques in program development. Firstly, programs are small and bounded-time, making them easier to verify efficiently than complex desktop programs. Secondly, security risks like those enumerated above combined with direct handling of money means that for smart contracts, correctness flaws are more likely to be security-critical and have greater direct financial impact than bugs in typical applications.

Lastly, contracts are encouraged to intercommunicate. This cross-contract communication paradigm has important consequences. Contract interactions are often complex, with safe contracts able to have their guarantees violated by calling into potentially malicious and unknown third party contract code. This paradigm further encourages reliance on purportedly safe "library" code supplied by professional developers. Unfortunately, many of these libraries are not verified formally or mechanically, and rely on human review to argue for their assurance[4]. These libraries of "standard" code are a good target for formal verification, but few tools exist for handling EVM programs, and even these tools may encode a model of EVM execution which diverges from the consensus-compatible implementations invoked by users.

With a massive recent uptick in corporate and industrial interest [1] as well as a skyrocketing marketcap and transaction count[5], the trend of high-profile and expensive losses in the Ethereum ecosystem appears likely to continue.

2.4 The $\mathbb{K}$ Framework

To perform our formalization of EVM, we employed the $\mathbb{K}$ Framework (`http://kframework.org`) [41]. We chose $\mathbb{K}$ for its language independent nature which emphasizes separating the semantics of a language from its related analysis tools, allowing for the development of an independent and readable reference semantics (which could even potentially replace the current semantic documentation in [45]).

[4] Example audit: `https://github.com/OpenZeppelin/zeppelin-solidity/blob/master/audit/ZeppelinAudit.md`

[5] In the year before this writing, Ethereum market capitalization grew from 820M USD to 26B USD, and daily transactions on the network increased from 40k to 300k according to `http://coinmarketcap.com/` and `http://etherscan.io/` respectively.

Other factors informing our choice of $\mathbb{K}$ included success in academia, where $\mathbb{K}$ has been used to formalize several real-world languages including C [15] [21], Java [4], and JavaScript [34]. Several language-independent analysis tools exist for $\mathbb{K}$; one such example is a program verifier based on symbolic execution of the language semantics [10]. $\mathbb{K}$ has also seen success in industry, providing the basis for a dynamic undefinedness checker and C analyzer shown to outperform popular static analysis tools on industry benchmarks [20] [12], rendering the approach practical.

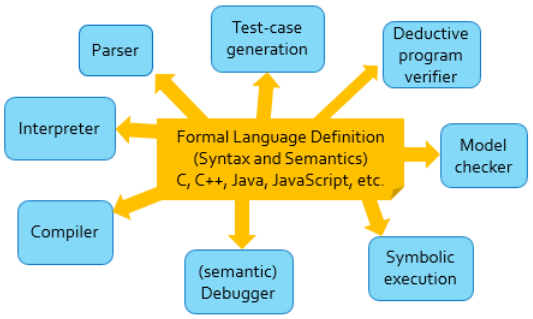

Figure 2: The $\mathbb{K}$ approach as described in and [10]

The foundation of $\mathbb{K}$ is Reachability Logic, a logic for reasoning symbolically about potentially infinite transition systems [9]. Reachability logic has a sound and relatively complete inference system which allows for efficient implementations. Versions of $\mathbb{K}$ are implementations of this logic, each able to reason about programs and systems. Currently, there are four versions of $\mathbb{K}$: K-Maude (the original prototype) [8], UIUC-K (for symbolic reasoning about programs) [10], RV-K (for fast concrete execution) [20], and Isabelle-K (for reasoning about $\mathbb{K}$ definitions in Isabelle) [24].

In this work, we present a $\mathbb{K}$ formal semantic definition of the Ethereum Virtual Machine (EVM) for use in verifying properties of smart contracts. Developing verification tools for each new target language is labor intensive and error-prone. $\mathbb{K}$ defines its tools parametrically over the input language, avoiding some of this development and maintenance cost. Parsers, interpreters, debuggers, and verifiers are generated directly from the formal definition (syntax and semantics) of the language, which is independent from the implementation details of these tools and auditable by the interested developer community.

As Figure 2 shows, $\mathbb{K}$ is a "semantics first" approach that emphasizes the development of a clear, complete, modular, and independent formal semantics of the target language and platform over the implementation of tool-level details. $\mathbb{K}$ provides several facilities for making language definition easier, including *configuration abstraction* and *local rewriting*, a technique allowing each transition rule to only mention relevant parts of execution state needed for that transition/rule.

Once a language is defined, $\mathbb{K}$ can read and execute programs in that language both on concrete and symbolic inputs, producing an interpreter and a symbolic execution engine, respectively. These can be regarded as formally derived reference implementations of the language, generated automatically from a rigorous semantics and usable for testing this semantics.

Our semantics of EVM in $\mathbb{K}$ lays the groundwork for rigorous and practical debugging, verification, and analysis of both EVM smart contracts and their interactions with the EVM network. Tools for checking EVM contracts for common bugs can be developed in a high-level yet semantically-rigorous way. This is immensely valuable to users of the EVM network, where fiscal risk means a clear semantics of the underlying platform is required and a wide range of competing implementations of the EVM running on the same network make a consistent semantics that lacks divergence or ambiguity in behaviors critical. In practice, divergences in implementation behavior have lead to persistent network splits between clients [36], a major failure of a consensus system.

3 $\mathbb{K}$ Semantics of EVM (KEVM)

Instantiating $\mathbb{K}$ with an EVM semantics requires modeling transaction execution state and network evolution using the *configuration*, and detailing changes in transaction execution and network evolution using transition *rules*. In this section, we outline the design of our KEVM semantics. We briefly overview the configuration of our semantics, and describe several example rules that provide insight into our approach. The KEVM semantics are open-source software and can be found at `https://github.com/kframework/evm-semantics`.

3.1 Logical Semantics Structure

KEVM is logically structured in three modules, each concerned with a different level of the semantics.

- **data.k** Data representations and their associated data structures used in the low-level EVM client code, and their definition in terms of $\mathbb{K}$-native data structures.
- **evm.k** Formalization of the EVM semantics in $\mathbb{K}$, including execution semantics of the various opcodes, world and network state, gas semantics, and various errors that can occur during execution. Implements the bulk of the Yellow Paper [45].
- **ethereum.k** Extra execution environment / drivers that run EVM code, with a mode to parse the JSON test-files used to test reference EVM implementations.

In the remainder of this section, we will focus on **evm.k**, and describe how some important or difficult EVM operations are formalized as $\mathbb{K}$ rules.

3.2 EVM Execution State (Configuration)

In $\mathbb{K}$, a *configuration* is used to specify latent state of the system so that the transition rules may easily access and update different components of that state. Each transition rule represents an execution step, and rewrites one configuration to another.

The state of EVM is broken into two components: that of an active transaction (smart contract execution), and the state of the network as a whole (account information). To mirror this, our configuration consists of two pieces. Only fragments of the entire configuration are shown here, as the actual configuration is sufficiently large to hold full state information.

Execution substate: Each execution of EVM has an associated account, a program counter, the current program, a word-stack, and a volatile local memory. Here is a hypothetical configuration mid-execution where the next opcode `MSTORE` will update locations `8` through `8 + 31` in the local memory to the value `75`. Several parts of the configuration are abstracted using ellipsis (`...`)[6].

```
       <op> MSTORE 8 75                 </op>
       <id> ...                         </id>
      <gas> 98422223                    </gas>
 <gasPrice> 1000000                     </gasPrice>
       <pc> 13                          </pc>
  <program> ... 13 |-> MSTORE ...       </program>
<wordStack> 7 : 9 : ...                 </wordStack>
 <localMem> ... 8 |-> _ ... 39 |-> _ ... </localMem>
```

The `<op>` cell holds the next execution step of the EVM and the `<id>` cell holds the ID of the currently executing account. The currently executing program is held as a map in the `<program>` cell, with the current program counter held in the `<pc>` cell. `<gas>` and `<gasPrice>` hold the available gas for execution and the amount of Ether the gas of this transaction costs respectively. The `<wordStack>` is the simple stack available to the EVM program being executed (implemented as a cons-list with operator `_:_`), and the `<localMem>` cell holds volatile auxiliary memory available to the program.

Network state: A blockchain (which is a log of state updates, consisting of groupings of transactions called blocks) corresponds to a set of account states.

[6]For the rules we show later in this section, ellipsis (`...`, called *structural frames*) are not omissions of details but actually syntax supported by $\mathbb{K}$ to avoid mentioning parts of the configuration.

```
<accounts>
    <account multiplicity="*" type="Map">
        <acctID>  ... </acctID>
        <balance> ... </balance>
        <code>    ... </code>
        <storage> ... </storage>
        <acctMap> ... </acctMap>
    </account>
</accounts>
```

The `<accounts>` cell holds information about the accounts on the blockchain. Each `<account>` holds the balance associated with the account in the `<balance>` cell, the program associated with it in the `<code>` cell, its permanent storage in the `<storage>` cell, and any other information about the account (like the account nonce) in the `<acctMap>` cell[7]. 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)[8].

3.3 EVM Hardware

KEVM provides the EVM execution cycle as a pipeline of commands over the current opcode, each one updating some part of the state. During this process, exceptions can be thrown and when caught might revert state. Here we describe this "hardware" that the EVM is then defined over.

Exceptional States: There are several points during VM execution where an exception may be thrown. For example, before attempting to execute an opcode we check if one of several exceptional cases will happen (formalized as the function `Z` in the Yellow Paper [45]):

1. Is the next opcode one of the designated invalid or undefined opcodes?
2. Will the stack over/under-flow upon executing the opcode?
3. If the opcode is a `JUMP*` code, will it jump to a valid destination?

```
    syntax InternalOp ::= "#exceptional?" "[" OpCode "]"
 // ----------------------------------------------------
    rule <op> #exceptional? [ OP ]
           => #invalid? [ OP ] ~> #stackNeeded? [ OP ] ~> #badJumpDest? [ OP ]
          ... </op>
```

The `=>` (pronounced "rewrites to") operator is supplied as a builtin in $\mathbb{K}$; rewriting is the transformation of one term to another, representing an execution step in the program [28]. In $\mathbb{K}$ definitions, `=>` can be used *locally*, meaning that it can show up at multiple points in a single rule (reducing the amount of extra state that must be specified). Rewriting is both the execution mechanism and the reasoning model in $\mathbb{K}$, with all KEVM rules and proof properties expressed as rewrites over the configuration.

The operator `_~>_` is also supplied with distributions of $\mathbb{K}$ and can be thought of as an associative sequencing operation (read as "followed by"). Here it means that to check `#exceptional?`, you must (in order) check `#invalid?`, then `#stackNeeded?`, and then `#badJumpDest?`.

The definition of the `#invalid? [ _ ]` check is fairly short, so we provide it here as an example. Notice here how if the `OP` satisfies the sort-check `isInvalidOp`, an `#exception` is generated.

[7] A nonce is a globally accessible monotonically increasing integer value that counts the number of transactions performed.

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

```
    syntax InvalidOp ::= "INVALID"
// ------------------------------

    syntax InternalOp ::= "#invalid?" "[" OpCode "]"
// -----------------------------------------------
    rule <op> #invalid? [ OP ] => #exception ... </op> requires isInvalidOp(OP)
    rule <op> #invalid? [ OP ] => . ... </op> requires notBool isInvalidOp(OP)
```

If the opcode is invalid, the check is replaced by `#exception` at the front of the `<op>` cell. Otherwise, the check is rewritten to `.` (the identity/unit of the "followed by" operator `_~>_`).

Control Flow: If any of the above exceptional checks fail, an exception is thrown (as demonstrated with the rules for `#invalid?`). The exception consumes any ordinary `Word` or `OpCode` behind it on the `<op>` cell, until it reaches a `KItem` (which it leaves).

```
    syntax KItem ::= Exception
    syntax Exception ::= "#exception"
// ---------------------------------
    rule <op> EX:Exception ~> (W:Word    => .) ... </op>
    rule <op> EX:Exception ~> (OP:OpCode => .) ... </op>
```

By putting operators in sort `KItem`, we can use them to affect how an `#exception` is caught. Here 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 <op>                #? K : _ ?# => K ... </op>
    rule <op> #exception ~> #? _ : K ?# => K ... </op>
```

Reverting State: During execution, state may need to be partially or fully reverted when an exception is thrown. To handle this, we supplied six operators `#{X}CallStack` and `#{X}WorldState` (with `{X}` one of `push`, `pop`, or `drop`). These operators can be used to save and restore copies of the execution and network state in the cells `<callStack>` and `<interimStates>`, respectively.

3.4 EVM Programs and Execution

EVM OpCodes: EVM programs are sequences of bytes corresponding to EVM opcodes; each opcode has an English mnemonic. The opcodes in EVM are broken into several subsorts of `OpCode` when common behavior can be associated to that group of opcodes.

```
syntax OpCode ::= NullStackOp | UnStackOp | BinStackOp | TernStackOp | QuadStackOp
                | InvalidOp | StackOp | InternalOp | CallOp | CallSixOp | PushOp
```

Argument Loading: When an opcode is executed, the correct number of arguments are unpacked from the `<wordStack>` (mostly determined by subsorting). The following rule simultaneously removes the top two elements of the `<wordStack>` and places them as arguments next to the `BinStackOp` (turning it into an `InternalOp` as the supplied production specifies). Note that here we're using local rewriting specification (there is one rewrite arrow `_=>_` per cell).

```
    syntax InternalOp ::= BinStackOp Word Word
// ----------------------------------------
    rule <op> #exec [ BOP:BinStackOp => BOP W0 W1 ] ... </op>
         <wordStack> W0 : W1 : WS => WS </wordStack>
```

Execution Cycle: The execution cycle for EVM first performs several checks for exceptional states, then actually performs the state update. Overall, it consists of three macro-steps:

1. Check if the opcode is exceptional given the current state (`#exceptional?`).
2. Perform the state update associated with the operator (`#exec`).
3. Increment the program counter to the correct value given the operator (`#pc`).

The following rule loads the next operator from the `<program>` cell using a structural map-match pattern (which means that the rest of the rule only applies if the lookup is successful). On successful lookup of `OP`, we save the current execution state with `#pushCallStack` then perform the three steps above. If any of these operations throws an `#exception`, it's caught with the choice operator `#?_:_?#` and the state is restored. Otherwise, the extra state on the `<callStack>` is dropped.

```
    rule <mode> EXECMODE </mode>
         <op> #next
           => #pushCallStack ~> #exceptional? [ OP ]
                             ~> #exec         [ OP ]
                             ~> #pc           [ OP ]
           ~> #? #dropCallStack : #popCallStack ~> #exception ?#
          ...
         </op>
         <pc> PCOUNT </pc>
         <program> ... PCOUNT |-> OP ... </program>
      requires EXECMODE in (SetItem(NORMAL) SetItem(VMTESTS))
```

Gas Semantics: The `#exec` operator above is turned into two consecutive state updates. First, the `#gas` update is applied (appendix G in the Yellow Paper [45]), then updates to execution state are applied (appendix H). As both computation and memory incur costs in the EVM model, to calculate `#gas`, we calculate the intrinsic gas due to execution followed by the cost of memory consumption.

```
    syntax InternalOp ::= "#gas" "[" OpCode "]" | "#deductGas"
// ----------------------------------------------------------
    rule <op> #gas [ OP ]
           => #gasExec(OP) ~> #memory(OP, MU) ~> #deductGas
          ... </op>
         <memoryUsed> MU </memoryUsed>
```

The execution gas calculation is done much as in the Yellow Paper [45]. Several fee schedules are provided (taken from the C++ implementation[9]); the desired schedule can be set with the command-line option `-cSCHEDULE=<FEE-SCHEDULE>`.

[9] `https://github.com/ethereum/cpp-ethereum/blob/develop/libevmcore/EVMSchedule.h`

```
    syntax Schedule ::= "DEFAULT" | "FRONTIER" | "HOMESTEAD" | ...
// ---------------------------------------------------------------
    rule <schedule> SCHED </schedule>
         <op> #gasExec(EXP W0 W1)
           => Gexp < SCHED > +Int (Gexpbyte < SCHED > *Int (1 +Int (log256Int(W1))))
          ... </op>
      requires W1 =/=K 0
```

The schedule constants are defined for each schedule:

```
    syntax Int ::= ScheduleConst "<" Schedule ">" [function]
    syntax ScheduleConst ::= ... | "Gexp" | "Gexpbyte" | ...
// ---------------------------------------------------------
    rule Gexp     < DEFAULT > => 10
    rule Gexpbyte < DEFAULT > => 10
    ...
    rule Gexpbyte < EIP158  > => 50
```

The more complicated gas calculations provide additional helper functions (eg. `Ccallgas` in the Yellow Paper [45]), which we faithfully implement in KEVM.

OpCode Semantics: Finally, after deducting the gas, we are ready to let the opcode execute on the world-state. Operator `#exec` places the opcode directly at the front of the `<op>` cell with its arguments from the `<wordStack>` loaded (as shown above).

For example, here we give the semantics of `ADD`:

```
    rule <op> ADD W0 W1 => W0 +Word W1 ~> #push ... </op>
```

`_+Word_` is a word operation supplied in the semantics which lifts the native $\mathbb{K}$ `_+Int_` operation and ensures the correct modulus-256 semantics are implemented for EVM. Behind the addition we place the helper `#push` which places the preceding word on the front of the `<wordStack>`.

Other opcodes require more of the state to operate. For example, memory operators `MLOAD` and `MSTORE` need access to the byte-array in `<localMem>`:

```
    syntax UnStackOp ::= "MLOAD"
// ----------------------------
    rule <op> MLOAD INDEX => #asWord(#range(LM, INDEX, 32)) ~> #push ... </op>
         <localMem> LM </localMem>

    syntax BinStackOp ::= "MSTORE"
// ------------------------------
    rule <op> MSTORE INDEX VALUE => . ... </op>
         <localMem> LM
                 => LM [ INDEX := #padToWidth(32, #asByteStack(VALUE)) ]
         </localMem>
```

Notice here how when we call `MLOAD`, we must read 32 contiguous bytes of data from the `<localMem>` and pack them into a single `Word` using `#asWord`. Similarly, when calling `MSTORE`, we must break the word up into 32 contiguous bytes (with `#asByteStack`) and then write them to a contiguous chunk of `<localMem>`.

Operator `BALANCE` needs to access the balance of the specified account:

```
    syntax UnStackOp ::= "BALANCE"
// ------------------------------
    rule <op> BALANCE ACCT => BAL ~> #push ... </op>
         <account>
           <acctID> ACCT </acctID>
           <balance> BAL </balance>
           ...
         </account>
```

Notice that we take advantage of $\mathbb{K}$'s *configuration abstraction*, which allows each rule to specify only the parts of the configuration relevant to its operation. As we have shown, this allows our rules to be concise, human readable, and capture precisely and exclusively the semantic nature of the relevant computation they define. This also makes semantic updates/maintenance simpler; changes in the structure of a cell only can affect the rules which mention that cell.

4 Semantics Evaluation

To evaluate our semantics, we focused on measuring *correctness*, *performance*, and *extensibility*.

As consensus-critical software, implementations of EVM are held to a high standard. Disagreeing on machine operation has caused accidental forks of the blockchain [36], so failing on the official test-suite is extremely consequential to the correct real-world operation of a blockchain system. Performance is secondary to correctness, but also important for real-world application and usability. Using the test set, we were able to collect measurements of our correctness and performance compared to selected other implementations.

Extensibility, the final property, is critical to ensuring that a formal semantics stays correct and updated as the underlying system evolves. Extensibility is not directly empirically measurable, so to establish the extensibility of KEVM we make a small addition, adding a monitor for gas analysis.

To reproduce our results on the test-set, first install and setup $\mathbb{K}$: `https://github.com/kframework/k` for UIUC-K and `https://github.com/runtimeverification/k` for RV-K. Then clone our repository at `https://github.com/kframework/evm-semantics` and follow the instructions there for setup and running the tests.

4.1 Correctness and Performance

We focus the concrete evaluation of KEVM on the official VMTests test suite released by the Ethereum foundation, available for download at `https://github.com/ethereum/tests/tree/develop/VMTests`.

Previous semantic work for the EVM, including the only currently available executable semantics [22], has used this test suite as the benchmark for semantic completeness. The test suite consists of 40,683 tests, over which KEVM passes all well-formed tests[10].

Table 2 shows a performance comparison between KEVM, the Lem semantics described in [22], and the C++ reference implementation distributed by the Ethereum foundation[11]. By comparing to the C++ reference implementation, we show the feasibility of using the KEVM formal semantics for prototyping, development, and test-suite evaluation.

All execution times are given as the full CPU time (negating the effects of parallelism, which some test harnesses are not designed to exploit). The evaluation machine was provisioned with an Intel i5-3330 workstation processor (3GHz on 4 hardware threads) with 24 GB of RAM.

Some measurements were not obtainable due to limitations in the test harnesses of the implementation making instrumentation without substantial modification difficult. Others were not obtainable for some implementations due to omissions in the provided test suite and corresponding test harness.

[10] We found one ill-formed test, which has been confirmed at `https://github.com/ethereum/tests/issues/216`.

[11] `https://github.com/ethereum/cpp-ethereum`

Test Set (no. tests)	**Lem EVM [22]**	**KEVM**	**cpp-ethereum**
Lem (40665)	288m39s (mean: 430ms)	34m13s (mean: 50ms)	3m6s (mean: 4.6ms)
Stress (18)	-	72m31s (mean: 240s)	2m25s (mean: 8.1s)
Normal (40665)	-	27m10s (mean: 40ms)	2m17s (mean: 3.4ms)
All (40683)	-	99m41s (mean: 150ms)	4m42s (mean: 6.9ms)

Table 2: Completeness and total execution time of existing executable semantics (- = unable to complete)

The row Lem indicates a run of all the tests that the Lem semantics can run. The median execution time for KEVM on the Lem tests is 36ms, well below the mean of 50ms, confirming that the stress-tests are skewing the measured execution time up. The row Stress indicates a run of all 18 stress tests in the test-suite, to compare the performance of KEVM with the C++ implementation. The row Normal is a run of all the non-stress tests in the test-suite (*not* the same set of tests as Lem). The last row, All, is the addition of the second and third rows (included for completeness).

As shown in the comparison, the automatically extracted interpreter for KEVM outperforms the currently available formal executable EVM semantics. In addition to handling the entirety of the test-suite (including those which the Lem semantics must skip), the KEVM semantics outperforms the Lem semantics by more than eight times. This performance speaks to the benefits of $\mathbb{K}$'s focus on executability of semantic specifications.

Compared to the C++ implementation KEVM performs favorably, coming in under 30 times slower on the stress tests, roughly 20 times slower on all the tests, and only 11 times slower on the Lem tests and the Normal tests. These numbers are very promising for an early version of an automatically generated, formally derived interpreter, and substantiates the practicality of our choice of approach.

Further improvements are added by using capable hardware; leveraging the parallelism inherent in the tests and a machine built for continuous integration (CI), we were able to run the full test suite (including stress tests) through KEVM in 7 minutes (excluding the stress-tests drops the time to 3 minutes). This positions KEVM as feasible for a high assurance CI environment; the same semantics used for language specification can reasonably be used for rapid testing.

We believe these promising empirical results support a bold long-term vision for KEVM: with a median transaction execution time of under 40ms and performance closing in on that of native reference implementations, our semantics is capable of handling millions of transactions per day on single-threaded commodity hardware, above the current daily volume of the Ethereum network. We believe that these results substantiate the future work direction of a fully formal reference client for Ethereum, in which the derived KEVM interpreter can process Ethereum transactions in real time, delivering the maximum possible assurance to users requiring it and complementing existing implementations designed for performance. Beyond running the tests during development/prototyping, the other tools derived from our semantics are practical for debugging both new features in the EVM and smart contracts written in EVM, as well as for formal analysis of smart contracts.

4.1.1 Yellow Paper and VMTests Deficiencies

Executable specification of programming languages is central to the $\mathbb{K}$ approach specifically because it means that testing is useful for determining completeness and correctness. The Yellow Paper [45], not being executable, cannot be tested for conformance to the other implementations; it only works as a reference specification as long as all changes to other implementations *first* change the Yellow Paper then their own code, an impractical approach when multiple implementations exist.

In addition to our performance and completeness evaluation, we attempted to debug the test suite itself by running a semantic coverage analysis over the provided tests. To do so, we simply tagged rules once they executed and gathered the set of rules never exercised by the test suite. This same technique was used on the semantics of KJS to uncover holes in the JavaScript test-set; when tests were added covering these semantic gaps bugs were found and fixed in many browser implemenatations [34].

In using the Yellow Paper as a guide for implementing KEVM, and the reference test suite as a validator, we uncovered several issues of importance to the Ethereum community. For example:

- There were no tests of `DELEGATECALL` (existing issue[12]).
- There were no tests of invalid opcodes (existing issue[13]).
- There is one test we reported as malformed (new issue with confirming response[14]).
- Many execution properties (e.g. the accumulated `refund`) are not exercised by the tests.

Deficiencies in the testset have important consequences for the network, as one of the test suite's primary functions is to expose potential divergences in implementation behavior before they result in accidental forks of the network [36]. As the EVM software evolves, these issues (and others) will be encountered again; the inefficiency of non-executable specification will grow. These results confirm the need for and validity of uncovering critical issues in EVM and other languages (and potentially therefore implementations) through rigorous, complete, and executable formalization.

4.2 Extensibility

We designed the semantics with extensibilty in mind. The simple imperative language (defined in Section 3.3) with control-flow supplied via exceptions and conditional branching allows us to add more primitives for extending KEVM. For example, we define another primitive `#execTo` which takes a set of opcodes and calls `#next` until we reach one of the opcodes in the set.

To have even more flexibility in the execution of the semantics, we make execution parametric over an extra `<mode>` cell[15]. Currently, three modes are supported: `NORMAL` execution, `VMTESTS` execution, and `GASANALYZE` mode.

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

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

The modes `NORMAL` and `VMTESTS` differ in that, when executing VMTests, implementations should only record the occurence of a `CALL*` or `CREATE` operation, and should not proceed to execute them. Mode `GASANALYZE` actually changes the execution cycle of `#next`, using the semantics to perform an analysis of the gas usage of programs. Any rule that should be dependent on the `<mode>` can simply add it to the cells mentioned in the rule, requiring it for execution of those rules.

4.2.1 Gas Analysis

EVM programs are forced to always terminate in order to prevent malicious actors on the network from DoS-attacking (Denial of Service) each other. This is done by allotting gas for execution ahead of time and having each step of execution consume some gas. If gas is exhausted before execution finishes, an exception is thrown and either a full revert or partial revert of state could happen. For many contracts, this means that their functional correctness is dependent upon enough gas being supplied at the beginning of execution. Unfortunately, the gas consumed during execution may be a function of the input, which has lead to losses including unrecoverable Ether stuck in contracts as described in [6].

To perform gas analysis of EVM programs, we augment the EVM semantics with a monitor which records gas and memory usage (storing results in the `<analysis>` cell). We perform a rather simplistic gas analysis; it will summarize the gas and memory used for each basic block of a program (delineated by `JUMP*` opcodes).

[12] `https://github.com/ethereum/tests/issues/136`
[13] `https://github.com/ethereum/tests/issues/160`
[14] `https://github.com/ethereum/tests/issues/216`
[15] The execution mode is set on the command line with `-cMODE=<EXECMODE>`.

Gas analysis leverages the `<mode>` cell by changing the behavior of `#next`. To begin, we open a basic-block summary (with `#beginSummary`) and call `#next`.

In gas analysis mode, `#next` checks if the next opcode is in the set `#gasBreaks` (consisting of `JUMP*` opcodes). If not, we will set the mode to `NORMAL` and execute until we do hit a `#gasBreaks` operator, then set the mode back to `GASANALYZE`.

```
rule <mode> GASANALYZE </mode>
     <op> #next
       => #setMode NORMAL ~> #execTo #gasBreaks ~> #setMode GASANALYZE
      ... </op>
     <pc> PCOUNT </pc>
     <program> ... PCOUNT |-> OP ... </program>
  requires notBool (OP in #gasBreaks)
```

When the next operator is in `#gasBreaks`, we perform several steps:

1. End the current summary (because we are at the end of a basic-block).
2. Move the program counter past the end of the basic block.
3. Set the gas back to a reasonably high value.
4. Begin a new basic-block summary.

```
rule <mode> GASANALYZE </mode>
     <op> #next
       => #endSummary
       ~> #setPC (PCOUNT +Int 1) ~> #setGas 1000000000
       ~> #beginSummary
      ... </op>
     <pc> PCOUNT </pc>
     <program> ... PCOUNT |-> OP ... </program>
  requires OP in #gasBreaks
```

Suppose we wanted to sum the numbers from `1` to `10` then store the result in the accounts storage. In the following psuedo-code, `sstore(0, s)` means "store the value `s` in location 0 of the current accounts memory", and is analogous to the EVM opcode `SSTORE`.

```
s = 0 ;
n = 10 ;
while (n > 0) {
    s = s + n ;
    n = n - 1 ;
}
sstore(0, s) ;
```

One implementation of this code in EVM (using the `<wordStack>` for storage/calculation), is as follows:

```
// s = 0; n = 10
      PUSH(1, 0) ; PUSH(1, 10)
// loop: if n == 0 jump to end
    ; JUMPDEST ; DUP(1) ; ISZERO ; PUSH(1, 21) ; JUMPI
// s = s + n;
    ; DUP(1) ; SWAP(2) ; ADD
```

```
// n = n - 1
    ; SWAP(1) ; PUSH(1, 1) ; SWAP(1) ; SUB
// jump to loop
    ; PUSH(1, 4) ; JUMP
// end: store to account
    ; JUMPDEST ; POP ; PUSH(1, 0) ; SSTORE
```

By calling `krun` with the option `-cMODE=GASANALYSE`, we produce the following output:

```
    ...
    <analysis> "blocks" |-> (
        ListItem ( { 0  ==> 4  | 6     | 0 } )      // s = 0; n = 10
        ListItem ( { 5  ==> 9  | 9     | 0 } )      // loop: if n == 0 jump to end
        ListItem ( { 10 ==> 20 | 24    | 0 } )      // s = s + n; n = n - 1;
        ListItem ( { 21 ==> 21 | 0     | 0 } )      // end:
        ListItem ( { 22 ==> 26 | 20005 | 0 } )      // store to account
    )
    </analysis>
    ...
```

This states that from program counter 0 to 4, 6 gas was used and the memory usage was not expanded. Similarly, from program counter position 22 to 26, 20005 gas was expended and the memory was expanded by 0 as well.

This (albeit simple) `GASANALYZE` mode only added 52 lines to the semantics (roughly 2% of the total length of the documented/literate semantics). Performance of the gas analyzer is also favorable: it only needs to execute each opcode of the program exactly once, and incurs only minor overhead in rules applied for tracking blocks. One can easily imagine more extensive analysis engines which provide upper and lower bounds of gas usage dependent on environmental parameters, or that take advantage of symbolic execution, leveraging the same basic gas-counting infrastructure.

The above transformations did not even directly encode the concrete cost of each operation, using what was already defined in KEVM. This demonstrates how further analysis tools can leverage rich semantic information that is already available in the modular, human readable semantics of KEVM to create formally rigorous tools that are also practical and easy to reason about. Notice that in the above example, a formally rigorous EVM tool was created while deferring execution details not relevant to the desired tool to the semantics. This brings the creation of formal semantics-driven tools into the realm of non-experts and removes the need for a separate per-tool model.

5 Derived Analysis Tools

Developing formal analysis tools for $\mathbb{K}$ definitions can be done in one of two ways: design and implement a language-independent analysis tool which then works over all $\mathbb{K}$ definitions, or augment an existing semantics with execution drivers/monitors which perform the analysis. Language-independent analysis tools can be time-consuming and expensive to develop, but yield payoff proportional to the number of languages with semantics in the framework. Language-specific semantic augmentations are comparatively simple to implement and can take advantage of assumptions made in the language itself, but may not generalize to other languages and may require tracking additional execution state.

$\mathbb{K}$'s language-agnostic framework encourages analysis tools to be developed independently of the programming language. This modular approach to formal analysis also ensures that the analysis tools are kept honest, in that they cannot have special hard-coded algorithmic decisions (potentially introducing bugs) for any specific language.

No new language-independent tools were developed for the EVM semantics; here we are presenting how the already existing language-independent $\mathbb{K}$ tools were applied to EVM program analysis. In this section, we

will demonstrate two tools derived from the formal semantics directly and automatically: the KEVM semantic debugger and KEVM program verifier. This presentation of is not meant to be exhaustive of the possibilities with our semantics or these tools.

5.1 Semantic Debugger

Adding the `--debugger` option to the command `krun` drops the user into the `KDebug` shell. In this shell, many debug commands are exposed to the user, including `step`, `peek`, and `back`[16]. These commands can be used to manually explore the execution of a target program given the semantics of the programming language. Note also that the semantic debugger is just a wrapper around the $\mathbb{K}$ symbolic execution engine, enabling it to handle symbolic states as well as concrete ones.

Our KEVM debugger proved enormously useful in defining the semantics of EVM, as when a test failed the debugger was able to step through the relevant portion of its execution to locate the failure precisely. Now that the semantics are complete and passing the tests, the debugger is still useful for analyzing individual EVM programs by stepping through their execution, providing rich semantic state information in addition to the traditional state information provided by a concrete debugger. This can be used to gain assurance in the behavior of an EVM program, and could even be integrated with traditional debugging tools for a further augmented debug environment.

Here we provide an example run of the semantic debugger, showing how it allows taking one step at a time through the program execution. For brevity, we have ommitted large parts of the configuration (with `...`). The number `N` in the following execution semantics is actually $2^{256} - 1$ in the concrete configuration of the program being executed.

Using `jump 91`, we skip to semantic execution step 91 (not corresponding to EVM program-counter 91). In this state, the top of the `<op>` cell has `#next`, indicating that the program counter should be used to lookup the next opcode to execute.

```
KDebug> jump 91
Jumped to Step Number 91
KDebug> peek
...
    <op> #next ~> #execute </op> ...
    <program> ... 33 |-> PUSH ( 32 , N ) ... </program> ...
    <wordStack> N : ... </wordStack> ...
    <pc> 33 </pc>
    <gas> 99997 </gas>
...
```

Taking one step, we see that a full execution cycle for the next opcode has been loaded. This consists of checking if the opcode is `#exceptional?`, executing the opcode with `#exec`, and incrementing the program counter with `#pc`. In this case, the opcode being executed is `PUSH(32, N)`, which is a 32-byte wide push of the number $N = 2^{256} - 1$ onto the top of the current VM stack.

```
KDebug> step
1 Step(s) Taken.
KDebug> peek
...
    <op> #pushCallStack ~> #exceptional? [ PUSH ( 32 , N ) ]
                        ~> #exec         [ PUSH ( 32 , N ) ]
                        ~> #pc           [ PUSH ( 32 , N ) ]
      ~> #? #dropCallStack : #popCallStack ~> #exception ?# ~> #execute </op> ...
```

[16] Press the Tab key for a list of commands.

```
    <program> ... </program> ...
    <wordStack> N : ... </wordStack> ...
    <pc> 33 </pc>
    <gas> 99997 </gas>
...
```

After 16 additional steps, we have fully executed the opcode (by processing all three steps of execution). The additional word has been pushed onto the `<wordStack>`, and the corresponding gas deduction has occurred in the `<gas>` cell. In addition, we see `#next` back on the `<op>` cell, indicating that the machine is ready for the next step of execution. The program counter has been advanced 33 positions, reflecting the fact that this was a 32-byte `PUSH` operation.[17]

```
KDebug> step 16
16 Step(s) Taken.
KDebug> peek
<generatedTop>
...
    <op> #next ~> #execute </op> ...
    <program> ... </program> ...
    <wordStack> N : ( N : ... ) </wordStack> ...
    <pc> 66 </pc>
    <gas> 99994 </gas>
...
```

The debugger is also helpful when trying to prove reachability claims about EVM programs (as in Section 5.2). Often it is necessary to supply an additional "loop-invariant" style reachability rule for reasoning about circular behavior during execution. Using the debugger, we can step to the front of the loop in question, copy the existing state, then step through the body of the loop once and copy the resulting state. While this pair of states cannot be used directly as a loop-invariant (some values may need to be generalized), they will fill in much of the reachability claim and guide the construction of a loop invariant and proof of the target property.

5.2 Program Verifier

One particularly useful formal analysis tool developed for $\mathbb{K}$ is the Reachability Logic prover [10]. This prover accepts as input a $\mathbb{K}$ definition and a set of logical reachability claims to prove. The prover then attempts to automatically prove the reachability theorems over the language's execution space, assuming the semantics. We now describe this process.

5.2.1 Reachability Logic

A reachability claim is a sentence of the form $\phi \Rightarrow \psi$, where ϕ are ψ are formulae in the static logic. In the case of $\mathbb{K}$, the static logic used is a fragment of Matching Logic [39]. The formulae of Matching Logic are called *patterns*, and can be thought of as configurations with symbolic variables for unknown values. Patterns can additionally contain constraints, meaning that a pattern can be thought of as the set of configurations which match the pattern structurally and satisfy all the constraints. Matching Logic allows representing code as data (algebraically), so the patterns ϕ and ψ can (and usually do) contain code.

The meaning of $\phi \Rightarrow \psi$ is that every state in the set of states represented by pattern ϕ will either reach a state in ψ or will not terminate, when executed with the given language semantics. In particular, a Hoare Logic triple $\{Pre\}Code\{Post\}$ can be written as the reachability claim $\overline{Code} \wedge \overline{Pre} \Rightarrow \epsilon \wedge \overline{Post}$, where ϵ is a

[17] `PUSH` is the sole operation in EVM which has its arguments inline in the program, meaning it does not increment the program counter by 1 like the rest of the opcodes.

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 :

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

Consequence :

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

Case Analysis :

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

Logic Framing :

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

Circularity :

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

Abstraction :

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

Figure 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). Interested readers should refer to [10] for a thorough presentation of using Reachability Logic to verify challenging properties of programs in real-world languages.

pattern representing the empty program. $\overline{Code}$ is a minimal state pattern containing the $Code$ but with selected program variables replaced with logical variables. Similarly for $\overline{Pre}$ and $\overline{Post}$ the variables are replaced with their logical counterparts; recall that in Hoare Logic no distinction between program and logical variables is made. A mechanical translation from Hoare Logic proofs to Reachability Logic proofs is worked out in [40], though we have found that it is often more intuitive to state reachability claims directly (as opposed to traditional Hoare triples).

Many interesting properties can be specified as a reachability claim, including all Hoare-style functional correctness claims. Verifying the complexity of a computation can be done in much the same manner as verifying functional correctness. In $\mathbb{K}$, an extra cell can be added to the *configuration* which increments each time one of the rules it is tracking is used. As an example, we will provide a rough sketch of verifying the number of integer multiplications in a program.

First add a cell `numMults` to the configuration counting the number of multiplications:

```
configuration ...
              <numMults> 0 </numMults>
              ...
```

Modify the rule which calls the builtin multiplication `_*Int_` to increment the counter:

```
rule <k> I *IExp I' => I *Int I' ... </k>
     <numMults> N => N +Int 1 </numMults>
```

Prove the following reachability theorem (where `f` is the "multiplication complexity" to prove):

```
rule <k> CODE => . </k>
     <numMults> N => f(N) </numMults>
```

If the reachability prover succeeds in generating a proof, it is the case that every program which starts

with `CODE` and `N` multiplications will perform an additional `f(N) - N` multiplications before terminating. This method can be used to track any complexity metric by transforming the associated rules (or summing across multiple associated rules).

In Hoare Logic, it is quite common that language-specific loop invariants must be supplied to aide in verifying programs with looping behavior. As discussed above, in Reachability Logic this is generalized to the language-independent notion of *circularity* (also stated as a reachability rule). Similar to Hoare Logic, the circularities must be supplied ahead of time (or inferred automatically, which $\mathbb{K}$ does not support). As the next section demonstrates, however, Reachability Logic circularities are often simpler to specify than Hoare Logic loop invariants. Instead of specifying an invariant in terms of the behavior of a single iteration of a loop, circularities can be specified as the behavior of the entire rest of the program after any execution of the loop.

5.2.2 Example Verification - Sum

We use the same sum example from Section 4.2.1 (denoted `PROGRAM`), except that `n` (10 in the program) starts with a symbolic value `N`. The property we verify is that the sum from 1 to `N` is calculated correctly by executing the program. This is expressed with the following reachability claim[18]:

```
rule <program> PROGRAM </program>
     <pc> 0 => 53 </pc>
     <wordStack> WS => 0 : N *Int (N +Int 1) /Int 2 : WS </wordStack>
     <gas> G => G -Int (52 *Int N +Int 27) </gas>
     <k> #execute ...</k>
     ...
requires N >=Int 0 andBool N <=Int 340282366920938463463374607431768211455
                                // ^^^ Largest N allowed to avoid overflow
 andBool #sizeWordStack(WS) <Int 1021
 andBool G >=Int 52 *Int N +Int 27
```

This states that starting at program counter 0, execution either diverges (does not terminate), or it passes through the state specified by the right-hand-side of the claim (this is the *relative completeness* of Reachability Logic [10]). Here, the right-hand-side states that we will reach program counter 53 with the correct sum calculated and stored as the second element of the `<wordStack>` (the top element being the final value of the loop counter). There are three (perhaps unexpected) constraints involving `N`, `#sizeWordStack(WS)`, and `G` (for gas). The extra constraints on `N` ensure that our sum remains low enough that integer overflow will not occur during execution, a critical condition for correct operation of this sum loop. Similarly, we must make sure that the size of the `<wordStack>` begins low enough that we will not trigger a stack overflow while executing this loop (EVM limits the `<wordStack>` to 1024 elements). Along the way, we also check a bound on the gas complexity (`G`) of the program by stating that it must begin high enough (`G >=Int 52 *Int N +Int 27`). As expected, the verification of this claim fails if any of these conditions is not included, indicating that any use of this code must check these conditions prior to execution to ensure correct behavior.

As the program has a loop starting at program counter 35, we need to supply a circularity which helps reason about how the remainder of the program behaves after any iteration of the loop. Note this is not quite the same as an invariant stating that each iteration of the loop maintains the partial sum (which one would have to specify using a Hoare Logic). Reachability Logic allows specifying the traditional Hoare Logic rule as a loop invariant and performing proofs that way, but specifying circularities directly instead can be more intuitive and robust to changes in the program. For example, the following rule only states that after any iteration of the loop, the remainder of the program is calculated correctly.

[18] Some cells of the configuration are omitted here with ellipsis for clarity.

```
rule <program> PROGRAM </program>
     <pc> 35 => 53 </pc>
     <wordStack> (I => 0) : (S => S +Int I *Int (I +Int 1) /Int 2) : WS:WordStack </wordStack>
     <gas> G => G -Int (52 *Int I +Int 21) </gas>
     <k> #execute ...</k>
     ...
 requires I >=Int 0 andBool S >=Int 0
  andBool S +Int I *Int (I +Int 1) /Int 2 <Int 2 ^Int 256
  andBool #sizeWordStack(WS) <Int 1021
  andBool G >=Int 52 *Int I +Int 21
```

`S` is the partial sum so far (counting down from the initial value `N`), and `I` is the next number to be added to the sum. This example contains the pre-condition `S +Int (I *Int (I +Int 1) /Int 2) <Int 2 ^Int 256` in the `requires` clause, which states that this code is not executed in a context where the sum would overflow.

Given these two reachability claims, our tool verifies that this EVM program correctly computes the sum from 1 to `N` as `N * (N + 1) / 2`, provided that all the required pre-conditions are met at execution time.

6 Case Study - ERC20 Token Standard

One major proof of concept for smart contract verification is the ERC20 token standard and the tokens that implement this standard. We chose this standard for verification due to its popularity in the Ethereum community as described in Section 2.3. 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 used to raise funding on the Ethereum platform [27]. This standard and its compliant tokens are therefore an attractive target for the practical impact of software verification, both in terms of financial impact and interest to the smart contract community as a whole. The ERC20 standard requires the following Solidity methods and events (and log items) to be implemented:

```
// Get the total token supply
function totalSupply() constant returns (uint256 totalSupply)
// Get the account balance of another account with address _owner
function balanceOf(address _owner) constant returns (uint256 balance)
// Send _value amount of tokens to address _to
function transfer(address _to, uint256 _value) returns (bool success)
// Send _value amount of tokens from address _from to address _to
function transferFrom(address _from, address _to, uint256 _value) returns (bool success)
// Allow _spender to withdraw from your account, multiple times, up to the _value amount.
// If this function is called again it overwrites the current allowance with _value.
function approve(address _spender, uint256 _value) returns (bool success)
// Returns the amount which _spender is still allowed to withdraw from _owner
function allowance(address _owner, address _spender) constant returns (uint256 remaining)
// Triggered when tokens are transferred.
event Transfer(address indexed _from, address indexed _to, uint256 _value)
// Triggered whenever approve(address _spender, uint256 _value) is called.
event Approval(address indexed _owner, address indexed _spender, uint256 _value)
```

This standard describes several simple methods that a smart contract must implement in order to be treated as a token by a variety of higher level applications (including wallets, exchanges, and other contracts expecting to interact with tokens). 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).

Ideally, we would like to create a specification describing generally desirable properties of ERC20, usable in verifying a wide range of implementations and therefore tokens at the EVM level. Performing this low-level verification also removes the need for compiler trust inherent with higher level language verification without a certifiable compiler. Such compiler problems are very real, and required the reissuance of a token holding and processing 190 million USD of value as the writing of this section was in progress. [19]

The Hacker Gold (HKG) Token Smart Contract: The HKG Token (an implementation of the ERC20 specification) was initially a topic of discussion when a vulnerability based in a typographical error lead to a reissuance of the entire token [26], disrupting a nontrivial economy based on it. Previously, the token was audited by human audit and deemed secure[20], further reinforcing the error-prone nature of the human review process and the need for tools that go beyond what is possible with manual review.

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 diferent, with the former being equivalent to a simple `=` (the second 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.

6.1 Procedure

We now describe the procedure we use to verify the `transferFrom` function of the HKG Token, which contained the vulnerable code requiring its reissuance[21]. This function is specified in the ERC20 standard described previously as "Send _value amount of tokens from address _from to address _to", and requires the `from` address to approve the transfer of at least the amount being sent using ERC20's approve functionality.

Initially, we were unable to verify the property over the vulnerable HKG code. Surprisingly, after fixing the bug which caused the reissuance, verifying against the ERC20 specification was still not possible due to the presence of an integer overflow bug not corrected in this reissuance. Additionally, because the KEVM semantics properly tests every condition which could result in an exception, we found that we must bound the remaining `<wordStack>` size to 1016 to avoid a stack overflow exception (as in Section 5.2.2). While a bounded stack size is a well known and documented limitation of the EVM, we did not explicitly reason about it during our initial proof attempts and were reminded to do so by the prover itself, further showing the power of full verification to find subtle cases in interactions between the contract and its underlying execution platform which may be missed by manual inspection. Supplied with these extra pre-conditions, the verification engine proved specified properties of the supplied ERC20.

The arithmetic overflow preconditions are potentially enforceable at runtime as runtime overflow checks (common in many more recently authored instantiations of ERC20 tokens), and with such checks, manually specifying these preconditions is unnecessary. The stack limits can be handled by either requiring the stack to have enough room at call time (shift reponsibility to the caller) or by allowing the function to throw if and only if the stack does *not* have room at call time, documenting the behavior of the method rigorously in a way that goes beyond the ERC20 specification.

This procedure represents the typical development / proving process for formal smart contract security, in which a contract and specification are codeveloped, revealing potential sources of error and false assumptions

[19] 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`.

[20] `https://zeppelin.solutions/security-audits`

[21] The relevant HKG source is available at `https://github.com/ether-camp/virtual-accelerator/blob/master/contracts/StandardToken.sol`.

until the complete proof states that the contract behaves as expected with regards to the desired property.

6.1.1 Compile Solidity Source To EVM

Since we are performing our verification at the EVM level, the first step in the verification process is to compile the HKG Token's source to EVM. To provide context for the verification process, we fixed the total token supply and added two dummy accounts with example balances before compiling the code to EVM. This simulates a series of transactions that would have occurred in the Ethereum world state before the verification of our property, and can be replaced by performing a series of state-updating deposit/token creation transactions.

Our example initial state contract describes this context:

```
contract StandardToken is TokenInterface {
//...
function StandardToken() {
    totalSupply = 5000;

    balances[dummy1] = 2000;
    balances[dummy2] = 3000;

    allowed[dummy1][dummy2] = balances[dummy1];
    allowed[dummy2][dummy1] = balances[dummy2];
}
```

Here is the Solidity code for the `transferFrom` function (with the `=+` bug fixed):

```
 /**
     * transferFrom() - used to move allowed funds from other owner
     *                  account
     *
     *  @param from  - move funds from account
     *  @param to    - move funds to account
     *  @param value - move the value
     *
     *  @return - return true on success false otherwise
     */
    function transferFrom(address from, address to, uint256 value) returns (bool success) {

        if ( balances[from] >= value &&
             allowed[from][msg.sender] >= value &&
             value > 0 ) {

            // do the actual transfer
            balances[from] -= value;
            balances[to] += value; // balances[to] =+ value in the vulnerable version

            // addjust the permision, after part of
            // permited to spend value was used
            allowed[from][msg.sender] -= value;

            // rise the Transfer event
            Transfer(from, to, value);
```

```
            return true;
        } else {
            return false;
        }
    }
}
```

6.1.2 Proof Claims

As explained in Section 5.2.1, the $\mathbb{K}$ prover takes proof obligations in the same format as rules from a language definition. Since our HKG Token contract contains only sequential code (no loops), our specification files (supplied via *-spec.k files) contain only a single claim each. The full claims are available online at `https://github.com/kframework/evm-semantics/tree/master/proofs`.

Here we summarize one of the claims, including interesting parts of the state (but ommitting 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 `Hacker Gold` token program, `TRANSFER` represents the symbolic amonut to transfer, `B1` and `B2` are the starting balances of accounts 1 and 2, repsectively, and `A1` is the allowance of account 1 (strictly smaller than the balance). The program counter starts at 818 and ends at 1331, which are the start and end of the `transferFrom` function in the compiled EVM.

```
rule <k>       #execute ...           </k>
     <id>      %ACCT_ID               </id>
     <program> %HKG_Program           </program>

     <pc>  818 => 1331         </pc>
     <gas> G   => G -Int 16071 </gas>

     <wordStack>                          TRANSFER : %CALLER_ID : %ORIGIN_ID : WS
              => A1 -Int TRANSFER : 0 : TRANSFER : %CALLER_ID : %ORIGIN_ID : WS
     </wordStack>

     <account>
       <acctID>  %ACCT_ID               </acctID>
       <code>    %function_transferFrom </code>
       <storage> %ACCT_1_BALANCE |-> (B1 => B1 -Int TRANSFER)
                 %ACCT_1_ALLOWED |-> (A1 => A1 -Int TRANSFER)
                 %ACCT_2_BALANCE |-> (B2 => B2 +Int TRANSFER)
                 %ACCT_2_ALLOWED |-> _
                 ...
       </storage>
       ...
     </account>
     ...
  requires TRANSFER >Int 0
   andBool B1 >=Int TRANSFER andBool B1               <Int 2 ^Int 256
   andBool B2 >=Int 0        andBool B2 +Int TRANSFER <Int 2 ^Int 256
   andBool A1 >=Int TRANSFER andBool A1               <Int 2 ^Int 256
   andBool #sizeWordStack(WS) <Int 1016
   andBool G >=Int 16071
```

The rule 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 `TRANSFER` from account 1 to account 2 (with `TRANSFER` sufficiently low and various overflow conditions

met) will happen as intended in the execution of the `transferFrom` code provided. This proves the desired property for the fixed version of the HKG contract, specifically that the storage balance entries are correctly updated in all possible cases of the execution of `transferFrom` in HKG, provided that all the arithmetic and stack overflow pre-conditions are met. While one can rely on the EVM to throw exceptions on stack overflow, arithmetic overflow will go undetected as the VM does not throw an exception on arithmetic overflow.

7 Related Work

There has been substantial practical interest in formally verifying properties of smart contracts for the reasons we enumerate in Section 2. Luu et al. [25] implemented a subset of the EVM, called EtherLite, and built a symbolic execution engine to check for common bugs in smart contracts; this has been developed into a commercially supported bug-finder/property-checker for EVM. The Solidity IDE incorporates Why3 [18] (a semi-automated theorem prover) to help verify smart contracts written in the higher-level Solidity language.

In this section, we will compare the practical and usable artifacts derived from and generated by our work to existing efforts in the space. We discuss both the various public implementations of a semantics of EVM and various practical software analysis tools for EVM programs. A list compiled recently by Dr. Yoichi Hirai[22] informs our comparison.

We do not include or compare with any tools which operate only over other languages (e.g., Solidity source analysis tools) or are exclusively implementations of an Ethereum client.

7.1 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. For the implementations we compare, we ask the following questions about the tools provided:

- **Spec.:** Suitable as a formal specification of the EVM language?
- **Exec.:** Executable on concrete tests?
- **Client:** Implements a full Ethereum client?
- **Verif.:** Used to verify properties about EVM programs?
- **Debug:** Supplies interactive EVM debugger?
- **Bugs:** Heuristic-based tools for finding common issues in smart contracts?
- **Gas:** Tools for analyzing gas complexity of an EVM program?

Table 3 shows an overview of the results of our comparison. We will now briefly describe each effort and compare it to the relevant artifact of our semantics.

7.2 EVM and Ethereum Tools

We separate our comparison projects into two categories: semantic specifications, strongly influencing and in some cases attempting to formalize EVM semantics, and smart contract analysis tools, intending to provide automated software quality tools for smart contracts.

7.2.1 Semantic Specifications

Because we include KEVM as a semantics specification, we perform a more in-depth comparison of it to the other semantic specifications available.

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

Tool	Spec.	Exec.	Client	Verif.	Debug	Bugs	Gas
Yellow Paper	✓	✗	✗	✗	✗	✗	✗
Lem semantics	✓	✓	✗	✓	✗	✗	✗
cpp-ethereum	✗	✓	✓	✗	✗	✗	✗
F⋆	✗	✓	✗	✓	✗	✓	✗
hsevm	✗	✓	✗	✗	✓	✗	✗
REMIX	✗	✓	✗	✗	✓	✗	✓
Oyente	✗	✓	✗	✓	✗	✓	✗
Dr. Y's	✗	✓	✗	✓	✓	✗	✗
KEVM	✓	✓	✗	✓	✓	✗	✓

Table 3: Feature comparison of EVM semantics and software quality tool efforts.

Yellow Paper: [45] The official document formally describing the execution of the EVM, as well as other required data, algorithms, and parameters required for building consensus-compatible EVM clients and Ethereum implementations. It is not executable, and thus not testable against the conformance test-suite, instead serving 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 for guidance.

cpp-ethereum: [23] 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, meaning that 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 will be favored. This implementation provides a full Ethereum client.

Lem semantics: [24] An Lem ([30]) implementation of EVM provides an executable semantics of EVM for doing formal verification of smart contracts. Lem is a language designed to compile 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 VM tests except for those dealing with more complicated intercontract execution, providing high levels of confidence in its correctness.

KEVM: Our implementation, described in this paper. Being written in a high-level rewriting-style language, our specification is human readable and can be used as a guide for implementations. Because KEVM is executable, it can be used as the reference implementation as well; this allows us to have a *single* specification to be used as both the formal model and the de-facto implementation semantics. The performance of KEVM suggests that it is even practical to integrate tools built directly from the semantics into the smart-contract development cycle, allowing for higher confidence in the smart contracts themselves. Finally, as rewriting specifications can quite naturally capture any non-determinism, our semantics needs no external reasoning about re-entrancy or non-determinism in EVM execution.

7.2.2 Smart Contract Analysis Tools

Compilation to F⋆: [3] An EVM analysis tool that compiles both Solidity and EVM programs to a functional language, F⋆. Functional correctness and runtime safety checks can be performed over the intermediate language

[23] `https://github.com/ethereum/cpp-ethereum`
[24] `https://github.com/pirapira/eth-isabelle`

F*. The translation serves as the semantics, which may be error-prone. As far as is stated in [3], the semantics has not been fully evaluated against the test suite.

hsevm: [25] A Haskell implementation of EVM focused on having a nice interactive debugger. It can be put into an interactive debug mode which allows stepping through contract execution one opcode at a time. Trust in its correct execution requires trust in the implementation.

REMIX: [26] A JavaScript implementation of the EVM with a browser-based IDE for building and debugging smart contracts. This tool can interact with a locally running Ethereum client to draw data from the main Ethereum network, and can interact with any node implementation (including potentially our formal semantics, if an RPC interface is added).

Oyente: [27] An EVM symbolic execution engine written in Python supporting most of the EVM opcodes. Built into the tool are several heuristics-based drivers of the symbolic execution engine for finding common bugs in EVM programs. These heuristics in many cases may contain false positives, and have not been rigorously proven to accurately capture their bug classes.

Dr.Y's Ethereum Contract Analyzer: [28] A symbolic execution engine for EVM to summarize the semantics of smart contracts. It comes with a debug mode where you can specify the number of execution steps and it will display possible contract states conditioned on inputs to the execution. Unlike our work, it makes no claim of being fully comprehensive in its coverage or faithful to the EVM semantics as defined in the implementation tests or Yellow Paper [45].

8 Future Work

We believe this full and rich ecosystem of tools, all generated programatically 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.

Semantic Completeness: Currently the VMTests (which test a wide range of VM functionality) are the only Ethereum tests being run against our definition. Soon we plan to also run the GeneralStateTests, which contain the pre-/post-conditions conditions for full transactions on the network instead of isolated chunks of EVM code. This will bring our implementation in line with the full features described in Section 7, eliminating remaining deficits of our work over other EVM implementations.

Analysis Tools: While we have made initial efforts towards the formal verification of smart contracts, there is still much work to be done to verify all of their nuanced properties. One key direction for future efforts is the formalization of classically known smart contract antipatterns, specifically those that have lead to security breaches and financial losses in the past. The definition of these antipatterns formally can then be used to generate a dynamic checker for EVM programs, similar to the dynamic checker for C in [12, 20].

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 one function (called `chop`). At execution time, if the function `chop` is called, the user will know their smart contract will have an overflow

[25] `https://github.com/dapphub/hsevm`
[26] `https://github.com/ethereum/remix`
[27] `https://github.com/melonproject/oyente`
[28] `https://blog.slock.it/an-ethereum-contract-analyzer-93e9da92fecb`

because execution will not finish. If a smart contract is verified correct in the modified semantics, it means that `chop` will never be called in the execution of programs which start in the pre-condition and execute to the post-condition.

Many simple bugs like integer overflow and stack over/underflow (demonstrated in Section 6.1) can be caught using this technique. As a good starting point, the contracts on website `http://hackthiscontract.io` can be used to inform how to modify the semantics to make these checkers. Each contract there contains subtle Solidity level bugs that users are encouraged to hack by crafting transactions which exploit the contracts' vulnerabilities. Tools to identify and warn about use of these antipatterns will help early in the smart contract development process, raising confidence in the final products.

ERC20: A further key extension of our work focuses on the completion of a full formal specification for ERC20, a popular smart contract specification in the Ethereum ecosystem. While we were able to verify many properties of each of the functions of an ERC20 token in isolation, the shim code inserted by the Solidity compiler to perform function dispatch and other tasks was not included in the analysis. Providing a fully verified reference ERC20 contract (including all ABI shim code) will be a great boon to the Ethereum community members attempting to write smart contracts.

High Level Languages: Another promising direction we intend to pursue is the formalization of higher-level smart contract development languages, including Serpent [16], Viper[29], and Solidity [17]. We expect this to be highly feasible, as these languages are either simple and well-defined (in the case of Viper) or inherit from existing languages heavily in their syntax and semantics (e.g., Solidity is similar to Javascript and Serpent is similar to Python). Both Javascript and Python have well-defined and relatively complete $\mathbb{K}$ Framework semantics, described in [34] and [19] respectively. Formalizing these smart contract languages therefore requires removing unnecessary constructs in the parent languages and adding EVM/smart contract-specific rules, many of which are already formalized in KEVM.

Underlying Language Paradigm: KEVM is a low-level bytecode-based assembly-like language. This makes it easy to force every computation to terminate (by charging some non-zero amount of gas for each opcode) and to agree on the proper execution of programs in the language. We suggest these alternative language paradigm choices for execution in the distributed computation setting:

- **Non-deterministic language:** Non-determinstic languages have many execution paths, but only some of the execution paths result in correct executions (solutions). This means that if you are supplied with the series of choices to make in a non-deterministic execution, verifying that they result in a solution is easier than finding that sequence of choices. Because EVM is deterministic, verifying the correct execution of a transaction takes as much effort as computing the correct execution of a transaction (namely, you must execute it!). Selecting a non-determinstic language may help with network scalability by ensuring that verifying a correct blockchain is easier than computing it the first time.
- **Language independence:** Most real-world computers are able to execute programs in many languages; a distributed execution environment should be no different. One approach to support many languages would be to compile them off-chain to the same underlying language - this is the approach that EVM takes. This can be made much simpler by switching out the underlying language for a language-building language; some contracts could be language specifications that future transactions can be written in. An added benefit of this approach is that compilation happens on-chain - everyone agrees on the correct way to interperet programs because the programming language specification is included in the blockchain.

[29] `https://github.com/ethereum/viper`

Each of these future directions of work, in isolation, will be useful to the Ethereum community. United by the common specification and verification infrastructure, though, these tools will act in concert to great effect. For example, formalizing a high level language as an extension of EVM along with analysis tools which check for EVM antipatterns will allow us to verify that compilers from the high level languages generate quality EVM code. The tools developed on top of KEVM are practical and performant enough for everyday use by software developers, but bring the power of automated formal techniques to bear as well.

9 Conclusion

Thus, we demonstrate a formalization of the EVM semantics in $\mathbb{K}$, providing an executable and human readable model of a reference semantics for EVM programs. This reference semantics can also be used to automatically generate a practical reference interpreter, which passes all the relevant official stress tests of compliant EVM interpretations. Our reference semantics is the first executable formal semantics able to make this claim of passing all official tests. We use this formalization exercise to uncover ambiguities in the Yellow Paper and potential gaps in test coverage, and better inform a rigorous understanding of the EVM execution model.

We assert and validate the computational practicality of our approach, showing execution results of KEVM's automatically generated EVM interpreter performing within an order of magnitude of native implementations on typical transactions, and within two orders of magnitudes on transactions specifically designed to stress test the EVM. We show that our semantics is practical for evaluation against a large number of test cases on commodity hardware today. We then demonstrate the extensibility of KEVM through example software analysis tools, able to be generated simply by monitoring relevant properties of interest in the semantics itself through the creation of a tool that computes gas bounds during execution.

We also show how, from this reference semantics, we are able to generate a wide range of analysis tools for EVM contracts automatically. We showcase an interactive semantics-based EVM debugger and a full program verifier that are automatically generated from our rigorous formal semantics, and demonstrate the verification of two example contract/property pairs. Lastly, we compare our work to other efforts in the space, showing KEVM to be the most comprehensive formally rigorous effort that provides both usable tools and an independently useful and complete formal semantic model.

By generating a wide range of tools from the same reference semantics, we reduce the probability of bugs arising from the encoding of these semantic models in existing software tools (many of which do not fully formalize the EVM). This further reinforces the utility of our "semantics first" development approach, in which software tools and reference implementations are generated automatically from a common and minimal reference semantics.

As interest in formal verification of EVM programs continues to grow, we hope that our contributions will pave the way for consistent and accurate verification of smart contracts, ultimately leading to a bug-free and secure development environment for the growing smart contract ecosystem.

9.1 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[30] 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)[31] 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

[30] `https://iohk.io/`

[31] `http://www.initc3.org/`

(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[32] and at Runtime Verification, Inc[33]. In particular, members Andrei Stefanescu, 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, and CNS-1514163, and an IOHK gift.

[32] `http://fsl.cs.illinois.edu`

[33] `https://runtimeverification.com/`

References

[1] Neil Ainger. Bigger than bitcoin? Enterprise Ethereum Alliance grows in size. 2017. http://www.cnbc.com/2017/05/23/bigger-than-bitcoin-enterprise-ethereum-alliance-grows-in-size.html.

[2] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on Ethereum smart contracts. *IACR Cryptology ePrint Archive*, 2016:1007, 2016.

[3] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, and Santiago Zanella-Béguelin. Formal verification of smart contracts: Short paper. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, PLAS '16, pages 91–96, New York, NY, USA, 2016. ACM.

[4] Denis Bogdănaş and Grigore Roşu. K-Java: A Complete Semantics of Java. In *Proceedings of the 42nd Symposium on Principles of Programming Languages (POPL'15)*, pages 445–456. ACM, January 2015.

[5] Lorenz Breidenbach, Phil Daian, Ari Juels, and Emin Gün Sirer. An in-depth look at the parity multisig bug. 2017. http://hackingdistributed.com/2017/07/22/deep-dive-parity-bug/.

[6] Vitalik Buterin. Thinking about smart contract security. 2016. https://blog.ethereum.org/2016/06/19/thinking-smart-contract-security/.

[7] Vitalik Buterin and Ethereum Foundation. Ethereum White Paper, 2013. https://github.com/ethereum/wiki/wiki/White-Paper.

[8] Traian Florin Şerbanuţă and Grigore Roşu. K-Maude: A rewriting based tool for semantics of programming languages. In *Proceedings of the 8th International Workshop on Rewriting Logic and Its Applications (WRLA'10)*, pages 104–122. Springer, March 2010.

[9] Andrei Ştefănescu, Ştefan Ciobâcă, Radu Mereuţă, Brandon M. Moore, Traian Florin Şerbănuţă, and Grigore Roşu. All-path reachability logic. In *Proceedings of the Joint 25th International Conference on Rewriting Techniques and Applications and 12th International Conference on Typed Lambda Calculi and Applications (RTA-TLCA'14)*, volume 8560 of *LNCS*, pages 425–440. Springer, July 2014.

[10] Andrei Ştefănescu, Daejun Park, Shijiao Yuwen, Yilong Li, and Grigore 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.

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

[12] Philip Daian, Dwight Guth, Chris Hathhorn, Yilong Li, Edgar Pek, Manasvi Saxena, Traian Florin Serbanuta, and Grigore Rosu. Runtime verification at work: A tutorial. In *Runtime Verification - 16th International Conference, RV 2016 Madrid, Spain, September 23-30, 2016, Proceedings*, volume 10012 of *Lecture Notes in Computer Science*, pages 46–67. Springer, September 2016.

[13] Michael del Castillo. Ethereum executes blockchain hard fork to return DAO funds. 2016. http://www.coindesk.com/ethereum-executes-blockchain-hard-fork-return-dao-investor-funds/.

[14] Kevin Delmolino, Mitchell Arnett, Ahmed Kosba, Andrew Miller, and Elaine Shi. Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab. In *International Conference on Financial Cryptography and Data Security*, pages 79–94. Springer, 2016.

[15] Chucky Ellison and Grigore 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)*, pages 533–544. ACM, January 2012.

[16] Ethereum. Ethereum serpent documentation, 2017. `https://github.com/ethereum/wiki/wiki/Serpent`.

[17] Ethereum. Ethereum solidity documentation, 2017. `https://solidity.readthedocs.io/en/develop/`.

[18] Jean-Christophe Filliâtre and Andrei Paskevich. *Why3 — Where Programs Meet Provers*, pages 125–128. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[19] Dwight Guth. A formal semantics of Python 3.3. 2013. `https://github.com/kframework/python-semantics`.

[20] Dwight Guth, Chris Hathhorn, Manasvi Saxena, and Grigore 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*, volume 9779 of *LNCS*, pages 447–453. Springer, July 2016.

[21] Chris Hathhorn, Chucky Ellison, and Grigore Roşu. Defining the undefinedness of c. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)*, pages 336–345. ACM, June 2015.

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

[23] Eleftherios Kokoris Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. Enhancing bitcoin security and performance with strong consistency via collective signing. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 279–296, Austin, TX, 2016. USENIX Association.

[24] Liyi Li. K Framework to Isabelle, 2015. `https://github.com/liyili2/KtoIsabelle`.

[25] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. Cryptology ePrint Archive, Report 2016/633, 2016. `http://eprint.iacr.org/2016/633`.

[26] Jim 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`.

[27] Steven Melendez. Inside the ico bubble: Why initial coin offerings have raised more than $1 billion since january. 2017. `https://www.fastcompany.com/40446051/inside-the-ico-bubble-why-initial-coin-offerings-have-raised-more-than-1-billion-since-january`.

[28] José Meseguer. Twenty years of rewriting logic. *The Journal of Logic and Algebraic Programming*, 81(7):721 – 781, 2012. Rewriting Logic and its Applications.

[29] Malte Moser, Rainer Bohme, and Dominic Breuker. An inquiry into money laundering tools in the bitcoin ecosystem. In *eCrime Researchers Summit (eCRS), 2013*, pages 1–14. IEEE, 2013.

[30] Dominic P Mulligan, Scott Owens, Kathryn E Gray, Tom Ridge, and Peter Sewell. Lem: reusable engineering of real-world semantics. In *ACM SIGPLAN Notices*, volume 49, pages 175–188. ACM, 2014.

[31] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008. `https://bitcoin.org/bitcoin.pdf`.

[32] Arvind Narayanan, Joseph Bonneau, Edward Felten, Andrew Miller, and Steven Goldfeder. *Bitcoin and Cryptocurrency Technologies: A Comprehensive Introduction*. Princeton University Press, 2016.

[33] Bobby Ong, Teik Ming Lee, Guo Li, and DLK Chuen. Evaluating the potential of alternative cryptocurrencies. *Handbook of digital currency. Amsterdam: Elsevier*, pages 81–135, 2015.

[34] Daejun Park, Andrei Ştefănescu, and Grigore Roşu. KJS: A complete formal semantics of JavaScript. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)*, pages 346–356. ACM, June 2015.

[35] Gareth W Peters and Efstathios 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*, pages 239–278. Springer, 2016.

[36] Andrew Quenston. Ethereum's blockchain accidentally splits. 2016. `https://www.cryptocoinsnews.com/ethereums-blockchain-accidentally-splits/`.

[37] Christian Reitwiessner. Dev update: Formal methods. 2016. `https://blog.ethereum.org/2016/09/01/formal-methods-roadmap/`.

[38] Pete Rizzo. In formal verification push, ethereum seeks smart contract certainty. 2016. `http://www.coindesk.com/ethereum-formal-verification-smart-contracts/`.

[39] Grigore Roşu. Matching logic. *Logical Methods in Computer Science*, to appear, 2017.

[40] Grigore Roşu and Andrei Ştefănescu. From Hoare Logic to Matching Logic Reachability. In *Proceedings of the 18th International Symposium on Formal Methods (FM'12)*, volume 7436 of *LNCS*, pages 387–402. Springer, Aug 2012.

[41] Grigore Roşu and Traian Florin Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010. `http://kframework.org/`.

[42] Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 459–474. IEEE, 2014.

[43] Nick 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`.

[44] Shawn Wilkinson, Tome Boshevski, Josh Brandoff, and Vitalik Buterin. Storj a peer-to-peer cloud storage network. 2014. `https://storj.io/storj.pdf`.

[45] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. 2014. (Updated for EIP-150 in 2017) `http://yellowpaper.io/`.