IELE: A Rigorously Designed Language and Tool Ecosystem for the Blockchain

Theodoros Kasampalis[1], Dwight Guth[2], Brandon Moore[2], Traian Florin Șerbănuță[2,3], Yi Zhang[1], Daniele Filaretti[2], Virgil Șerbănuță[2], Ralph Johnson[2], and Grigore Roșu[1,2]

[1] University of Illinois at Urbana-Champaign
[2] Runtime Verification
[3] University of Bucharest

Abstract. This paper proposes IELE, an LLVM-style language, together with a tool ecosystem for implementing and formally reasoning about smart contracts on the blockchain. IELE was designed by specifying its semantics formally in the K framework. Its implementation, a IELE virtual machine (VM), as well as a formal verification tool for IELE smart contracts, were automatically generated from the formal specification. The automatically generated formal verification tool allows us to formally verify smart contracts without any gap between the verifier and the actual VM. A compiler from Solidity, the predominant high-level language for smart contracts, to IELE has also been (manually) implemented, so Ethereum contracts can now also be executed on IELE.

1 Introduction

Ethereum [5], with more that 500,000 daily transactions [15] is the largest blockchain network supporting smart contracts. The smart contracts used in the context of Ethereum transactions are written in the language of the Ethereum Virtual Machine (EVM) [44], a stack-based assembly-level language.

Unfortunately, recent exploits on EVM smart contracts have led to losses in the range of hundreds of millions USD [1,8,41,4,40]. In response, KEVM [19] was developed, a formal semantics of the EVM in K [34], to provide formal verification assistance to the EVM smart contract world [31,37]. We and others in the Ethereum community have embraced and adopted KEVM, as a more rigorous and thus precise alternative to the Yellow paper [44]. Through our own experience with KEVM and the reports of others, we became aware of the limitations of EVM as a language: it includes features that are easily exploitable and its low-level nature makes formal verification efforts tedious and time-consuming. On the positive side, as reported in [19], the EVM interpreter automatically generated from KEVM by the K framework [34] was only one order of magnitude slower than the official EVM client written in C++ [11]. Since the node client code takes only a small fraction of the total execution time on a blockchain, the KEVM performance was considered acceptable by IOHK (http://iohk.io), the company that is in charge of the Ethereum Classic blockchain, to deploy a testnet that is entirely powered by the auto-generated KEVM client [21].

In response to these limitations of EVM, we have designed and implemented IELE, a new language for smart contracts. IELE is a low-level language with syntax similar to that of LLVM [26]. It is designed to be both human readable and suitable as a compilation target for more high-level languages. IELE has various high-level features, such as function calls/returns, static jumps, arbitrary-precision integer arithmetic among others, that both make automatic formal verification more straight-forward and the language itself more secure.

IELE was designed using a formal specification, and its implementation was automatically generated from its specification using the same technology that was used to generate the implementation of KEVM [19], namely K [34]. In contrast, other languages have separate specifications and implementations, and it is hard to keep them from differing. Sometimes the specification is informal, like that of LLVM [26] and sometimes formal, like that of SML [27], but in either case the implementation is separate so it is not possible to execute test cases against the specification. To bridge the gap between specification and tools, K provides support for developing language semantic definitions as well as a host of tools for such a definition, such as parser, interpreter, deductive verifier, and more. These tools are automatically generated so any change to the formal semantics is automatically propagated to the tools. For IELE, we use the generated parser/interpreter to obtain a IELE Virtual Machine (IELE VM) tool (also reference implementation). We also use the capability of K to generate sound and relatively complete deductive program verifiers for the defined languages [7], to obtain a IELE smart contract formal verification tool. As a result, IELE is formally specified and its implementation and verifier are correct by construction and remain correct, with zero effort, in the presence of updates to the language.

We have deployed the IELE VM in a testnet supported by IOHK, a major blockchain company [20]. To do so, we have built appropriate infrastructure around the IELE VM: a full compiler for Solidity [12], a popular high-level language for smart contracts that until now could only be compiled to EVM, as well as integration with the Mantis Ethereum client [22]. This allows us to run real-world Solidity contracts on IELE. Appendix B discusses this effort in detail.

In summary, our contributions are:

- IELE, a smart contract language designed from the ground up using formal methods with the goals of security, verification, human-readability, and portability in mind, using the K framework. IELE is publicly available [38].
- Useful tools for IELE: A IELE virtual machine that was deployed as a testnet by a major blockchain company and a IELE smart contract verifier automatically generated by the K framework from the IELE formal semantics.

More details can be found on Github [38] and on the IELE testnet [20].

2 Background

We briefly discuss the K framework, Ethereum and the EVM, as well as other smart contract languages currently under development.

The K Language Semantic Framework K [34] is a rewriting-/reachability-based framework for defining executable semantic specifications of programming

languages. Given the syntax and semantics of a language, K automatically generates a parser, an interpreter, as well as formal analysis tools such as a deductive verifier. This avoids duplication while improving efficiency and consistency. For example, using the interpreter, one can test the semantics immediately, which significantly increases the efficiency of and confidence in semantics development. There exists a rich literature on using K for formalizing existing languages, such as C [10,18], Java [3] and JavaScript [30], among others. K has also been used to formally specify the Ethereum Virtual Machine (EVM) [19], the current smart contract language for Ethereum. In fact, the process of formalizing EVM as an executable semantics uncovered various inconsistencies and unspecified behaviors in its original English specification (the Yellow paper) [44].

Blockchain and Ethereum A blockchain is an append-only ledger that is commonly used for synchronization in distributed protocols. Cryptocurrencies such as Bitcoin [29], refer to such protocols that allow a set of clients to transfer and maintain a balance of virtual coins. A cryptocurrency network consists of accounts (essentially encrypted client IDs) with cryptocurrency balances, a blockchain of verified transactions, and a set of so-called miners, computation resources that process pending transactions and append them in the blockchain.

Ethereum [5] is a blockchain-based network that provides a decentralized, replicated computer for distributed applications and uses a blockchain to store its global state. Ethereum supports a programming language for writing *smart contracts*, which are programs associated with Ethereum accounts. Ethereum accounts interact with each other using transactions over a cryptocurrency called Ether. When an Ethereum account associated with a smart contract receives a transaction, execution of its smart contract code is triggered. Such transactions can both transfer Ether and pass input data to the smart contract.

Ethereum smart contracts often manage large monetary amounts, in the range of 100M USD. Ethereum's popularity is largely due to the fact that there is no need for a trusted third party (such as a bank) to verify the transactions; the trust comes from the consensus algorithm and the fact that smart contract code is binding. Ethereum transactions are irreversible and the source code of the involved smart contracts is public and immutable. Moreover, any transaction can be replicated through the information stored in the blockchain.

EVM Currently, Ethereum contracts should be translated to the language of the Ethereum Virtual Machine (EVM) [44]. The EVM is a stack-based VM with an assembly-level language with no code/data separation or function-level calls/returns. The language is loosely specified in the Yellow Paper [44], but has been recently formally specified as a K definition in KEVM [19]. KEVM provides a reference implementation that is considered for adoption by the Ethereum Foundation, a deductive verifier that has been used to verify real-world contracts [31,37], and the Jello Paper [25], an English language specification of EVM generated from the documentation of the KEVM semantics definition.

Other Smart Contract Languages Several smart contract programming languages have been proposed either as alternatives to EVM or as higher-level, more programmer-friendly options. Solidity [12] and Vyper [14] are popular

high-level languages for Ethereum contracts that are typically compiled down to EVM. Plutus [23] is a high-level functional language that offers increased security due to features such as type safety. Rholang [33] is a functional, concurrency-oriented language that powers RChain, an evolution of traditional blockchain networks that allows for concurrent transactions. We believe that IELE could serve as a compilation target of all these languages. In fact, we have implemented a Solidity-to-IELE compiler as part of our evaluation (see appendix B.1). Intermediate-level smart contract languages include Michelson [42], a stack-based, but also statically typed language, Scilla [45], a language that offers clear separation between in-contract computation and inter-contract communication to facilitate formal reasoning about smart contracts, and Simplicity [2], a language designed to have simple semantics which lend themselves to static analysis and formal reasoning. IELE shares the goals of enhanced security and ease of formal verification with these languages. We believe that the novelty in the design of IELE, compared to those efforts, stems from the fact that its formal specification and its de-facto implementation are one and the same artifact: any change to IELE's specification is automatically propagated to the implementation of the language execution engine and verification tools.

3 The IELE language

Based on experience with KEVM and formal verification of EVM smart contracts, we identified five desired properties for an ideal blockchain low-level language and designed IELE around them. Our design was done using formal semantics, and the implementation was generated automatically [38,20].

Security Smart contracts often manage large monetary amounts and have been targets of attackers that seek to exploit any vulnerabilities in their code [1]. Very often, language design weaknesses such as undefined behaviors, execution of arbitrary data as code, and silent integer overflow act as enablers for attackers to exploit corresponding bugs. IELE avoids all of the above (and more) design weaknesses and hence eliminates many possible attack vectors by design.

Formal verification No matter how many insecure features are avoided at the language definition level, software bugs can always allow for exploits. The three most expensive exploits of Ethereum smart contracts are all due to software errors [8,4,41]. A strong defense against such exploits is formal correctness verification and IELE is designed with the goal of formal verification in mind.

Human readable Smart contracts act as binding agreements between human end-users. Being human readable reinforces this intention, as it is easier for the agreeing parties to trust a formal agreement they can read and understand. Ideally smart contracts should be human readable at the exact level they are stored in the blockchain and executed. This is true for IELE contracts, since the IELE syntax was designed to be almost identical to that of LLVM [26], a state-of-the-art intermediate language designed to be human readable.

Determinism Ethereum's blockchain stores transactions that can be replicated by any Ethereum client and many of these transactions require execution of smart contract code to be replayed. For this reason it is important that the

underlying smart contract language and its implementation are deterministic. The IELE specification contains no undefined and/or implementation-defined behaviors, as well as no by-design non-deterministic features.

Gas model The philosophy behind IELE's gas model is simple: no limitations in code execution, but costs are analogous to the resource consumption. For example, IELE programs have access to an unlimited number of registers, but more used registers incur steeper gas charges. Similarly IELE uses unbounded integer arithmetic, but the larger the numbers at runtime, the more gas required. A more detailed discussion of IELE's gas model can be found in appendix A.

Listing 1 shows a simple forwarder contract in IELE. The contract forwards any amount of Ether[4] sent to it to the account that created the contract. The `@init` function of the contract is executed when a transaction creates this contract. The built-in `@iele.caller` returns the account address of the account that posted the transaction, which is the creator of this contract. This address is saved in the account storage of the forwarder. The `@deposit` function is a public function meaning it can be invoked by incoming transactions. The built-in `@iele.callvalue` returns the amount of Ether that was sent to the forwarder with the incoming transaction. This amount is forwarded to the creator account by invoking its own `@deposit` function with the IELE instruction `call .. at`. Note that the forwarder specifies an upper limit of gas to spend at the creator during the account call. If the forwarding fails (e.g., due to lack of gas), the built-in `@iele.invalid` is called that reverts any global state change made so far, including the Ether receipt for the forwarder account.

The full formal semantics of the IELE language, given as an executable K definition that also serves as a reference implementation, can be found in [38]. In the remainder of this section we give a high-level presentation of IELE and discuss the improvements over the state-of-the-art EVM.

3.1 IELE Contracts

A IELE contract is the main compilation unit of code that can be associated with a blockchain account. A IELE contract has a name and contains one or more functions, global variables, and external contract declarations. Public functions can be invoked from other accounts while private functions only from within the contract. Global variables are accessible from anywhere within the contract and hold a constant value. The Listing 1 contract has one global variable, `@creator`, one private function, `@init`, and one public function, `@deposit`.

Account Storage An account includes a storage that is an unbounded sparse array of arbitrary-precision signed integers. The storage is persistent, i.e., it holds its contents throughout the account's lifetime. As such the storage contents of all accounts are part of the global state and any modification on them is recorded in the blockchain. IELE code associated with an account can access the account storage through the dedicated `sload` and `sstore` instructions. IELE global variables are typically used to hold specific storage addresses, so that the

[4] IELE can be used in any Ethereum-style blockchain, where the cryptocurrency may be called differently. We use Ether throughout the paper for the sake of concreteness.

contract code can refer to those addresses by the name of the variable. See, e.g., how the global variable `@creator` is used in the contract of Listing 1.

Listing 1. Forwarder contract in IELE

```
// Contract forwards any funds it receives to its creator account
contract Forwarder {
  // account storage index holding the creator account address
  @creator = 1
  // initializes a forwarder
  define @init() {
    // get the address of the creator account
    %creator = call @iele.caller()
    // store the creator's address in the storage
    sstore %creator, @creator
  }
  // forwards the received funds to the creator of this account
  define public @deposit() {
    // get the received funds
    %value = call @iele.callvalue()
    // get the creator account address from the storage
    %creator = sload @creator
    // forward funds by calling deposit at the creator account
    %status = call @deposit at %creator () send %value ,
              gaslimit 2300
    br %status, throw // contract call failed
    ret void
  throw:
    call @iele.invalid()
  }
}
```

Contract Creation An account can be created and associated with a IELE contract manually by an end user, by posting of an appropriate transaction, or dynamically by another executing IELE contract, using IELE's `create` instruction. We say that a new contract created this way is deployed, since an executable object (including smart contract code and state) is literally stored in the blockchain. Listing 2 shows a contract that dynamically creates accounts associated with the `Forwarder` contract shown in Listing 1. The `create` instruction dynamically creates a new account, deploys the Forwarder smart contract, and finally returns the new account's address. The `send` attribute specifies that no initial amount of Ether is sent to the new account.

The smart contract code to be associated with a newly created account should be available at creation time. In case of dynamic account creation, we chose to design IELE with the stricter requirement that the code should have been available at creation time of the creator contract. For this purpose, a IELE contract contains a list of external contracts that it is allowed to create at runtime, and the code for each of these contracts should be available when the contract itself is created. Recursively, the code of each contract that these contracts may create should also be available. Hence, all code that can be stored in the blockchain is available at the time when some end user posts an account creation transaction.

In Listing 2, the `Wallet` contract declares the `Forwarder` contract as external. This means that if a new `Wallet` contract is to be created and deployed, at that time both the code of the `Wallet` contract and the code of the `Forwarder` contract should be provided and will be stored in the blockchain as part of the

creation transaction. Later, during execution, the `Wallet` contract is able to dynamically create `Forwarder` contracts using the available `Forwarder` code.

Listing 2. Wallet contract that creates Forwarder contracts

```
// Wallet contract that creates new accounts that forward funds sent to them to this wallet.
contract Wallet {
  external contract Forwarder
  define @init() {}

  // @deposit is empty: any received funds are simply added the associated account's balance
  define public @deposit() {}

  // creates a new account that simply forwards any funds sent to it to this wallet and
  // returns the address of the created forwarder account
  define public @newForwarder() {
    // ensure that the caller is the account associated with the wallet
    %caller = call @iele.caller()
    %owner = call @iele.address()
    %isnotowner = cmp ne %caller, %owner
    br %isnotowner, throw
    // create a new account associated with a Forwarder contract and return its address
    %status, %addr = create Forwarder () send 0
    br %status, throw
    ret %addr
  throw:
    call @iele.invalid()
  }
}
```

Dynamic contract creation is thus guaranteed to only use code that has already become available in the blockchain; *no dynamic code generation is allowed*. This design has two major advantages. Expensive code validation checks (well-formedness, formal verification, etc.) need only take place when account creation transactions are posted and never during code execution. Also code can be stored in the blockchain separately from the account it is associated with: contracts can be stored in a separate storage (with no duplicate contract code) and accounts need only store a pointer to their associated code. This allows for cheap dynamic account creation that doesn't generate duplicate code in the blockchain.

Contract Initialization A special private `@init` function can be defined and will be executed at contract creation time. This function typically contains initialization code, e.g. initialization of the account storage. It is not callable and it can only be invoked at contract creation time. This way, IELE guarantees that the state of an already deployed contract cannot be reset maliciously by invoking initialization code after contract creation, thus avoiding a weakness of the current Ethereum design that has been exploited in the past [4].

3.2 IELE Functions

IELE functions are the main structural units of a IELE contract. A function definition includes the function signature, the function body and whether or not the function is public. A function signature includes a function name and names of formal arguments. A public function can be called by other accounts, while a private one can only be called by other functions within the same contract. Listing 3 shows a simple implementation of the factorial as a IELE function.

Control Flow The function body code is organized in labeled blocks. The execution falls through from the last instruction of a block to the first of the

Listing 3. IELE function for factorial

```
define public @factorial(%n) {
  // ensure %n >= 0
  %lt = cmp lt %n, 0
  br %lt, throw
  %result = 1
condition:
  %cond = cmp le %n, 0
  br %cond, after_loop
loop_body:
  %result = mul %result, %n
  %n      = sub %n, 1
  br condition
after_loop:
  ret %result
throw:
  call @iele.invalid()
}
```

next one, or jumps to the start of a specific block. IELE supports jumps to statically known targets only: The branch instruction accepts a block label as an argument for the target of the jump. This differs from the EVM, where jumps amount to pushing a possibly computed number on the stack and then jumping to it regarded as an address. IELE ensures a statically known control flow graph and thus makes static analysis and formal verification easier.

Function Calls A public function can be invoked manually by an end user posting a transaction, or dynamically by another executing contract using IELE's account call instruction, `call .. at`. The address of the callee and the name of the called function are provided at call time. The call may be accompanied with an Ether amount to be transferred from the caller to the callee. IELE defines a simple call/return convention: Called functions expect a specific number of arguments (the number of formal arguments) and return a specific number of return values (the number of return values at `ret` plus an exit status). If an ABI error occurs (e.g. incorrect number of arguments, function not found or not public, etc), a corresponding erroneous exit status is returned.

For reference, in EVM the caller just sends an arbitrary bytestream containing the call arguments. There is an externally defined ABI convention that most EVM compilers follow, but it is not enforced. EVM does not have a notion of callable function; instead, the execution always starts at the contract's first instruction. The higher abstraction level of IELE's calls and its more structured design makes IELE contracts more readable and less tedious to formally verify.

EVM supports another type of account call, namely `delegatecall`, that differs from the normal call, in that the code of the callee is executed in the environment of the caller. This essentially means that the storage of the caller account becomes accessible and writable from the callee code. EVM offers this feature to avoid code duplication: typically library code is associated with a single account and invoked with `delegatecall` by multiple clients. IELE offers a different solution to the code duplication problem as discussed in Section 3.1. For this reason and because `delegatecall` poses serious security concerns and has been exploited in the past [41], we decided to drop `delegatecall` in IELE.

Deposit Handler A special public function, `@deposit`, can be defined for a IELE contract and it is invoked whenever the account receives a payment

that is not accompanied with a specific function call. A contract can forbid such payments by refusing to define a `@deposit` function.

3.3 IELE Instructions

IELE instructions take the form of opcodes that accept a specific number of arguments and return a specific number of values. There are various families of instructions, including arithmetic, bitwise, comparison, and hashing operations. There are also dedicated instructions for accessing the local memory, accessing the account storage, and appending to the account log. Finally there are branch instructions, the function call/return instructions, and the account create/selfdestruct instructions. In addition to instructions, IELE supports a number of useful intrinsics that can be called like private functions and provide functionality such as querying the local or network state, cryptographic functions, etc.

IELE is register-based: Instructions operate on and store their output in virtual registers. An infinite number of virtual registers is available, but the actual number of registers used by the function can be determined statically by counting the different register names used in its code. We chose to design IELE as a register-based language, unlike the stack-based EVM, for two reasons. First, it makes IELE code significantly more human-readable. Second, IELE formal verification tools do not need to reason about the size of the operation stack (bounded to 1024 words, a tedious requirement for verifying EVM programs).

3.4 IELE Datatypes

IELE uses arbitrary-precision signed integers. All virtual registers and account storage locations hold values of this type. They can also be stored in and loaded from the local memory. Arbitrary-precision arithmetic removes arithmetic overflows thus making specification and formal verification less tedious, as well as making attacks like [32] that exploit arithmetic overflow bugs not viable.

4 Formal IELE Language Definition in K

Here we describe the formal semantics of IELE, from which several of our IELE tools are generated automatically by the K framework, such as an interpreter, a well-formedness checker, and a program verifier. In total, the IELE semantics consists of 3122 lines of K code (not including literate comments), including 729 productions (for syntactic or semantic constructs) and 1255 semantic rules [38]. For comparison, the K EVM semantics in [19] consists of 2628 lines of K code, including 510 productions and 1025 semantic rules; these total less than 20% the size of the VM code component of the official EVM client implemented in C++ (about 15k LOC) [11]. A similar save would be seen if IELE had a conventional VM implementation in C++, although such an implementation was never needed because the IELE VM was automatically generated from the formal semantics.

4.1 IELE Formal Semantics Overview

The formal semantics of IELE [38] is spread among several files: `iele-syntax.md` contains a quick introduction to the various features of the language along with

Listing 4. K syntax productions for IELE contracts

```
syntax ContractDefinition ::= "contract" IeleName "{" TopLevelDefinitions "}"
syntax TopLevelDefinitions ::= List{TopLevelDefinition, ""}
syntax TopLevelDefinition ::= FunctionDefinition
                            | GlobalDefinition
                            | ContractDeclaration
syntax GlobalDefinition ::= GlobalName "=" IntConstant
syntax ContractDeclaration ::= "external" "contract" IeleName
syntax FunctionDefinition ::= "define" FunctionSignature "{" Blocks "}"
                            | "define" "public" FunctionSignature "{" Blocks "}"
syntax FunctionSignature ::= GlobalName "(" FunctionParameters ")"
syntax FunctionParameters ::= LocalNames
syntax GlobalName ::= "@" IeleName
syntax LocalName ::= "%" IeleName
syntax LocalNames ::= List{LocalName, ","}
```

their syntactical definitions. `iele.md` contains the semantics of the various language features and the specification of the program execution state. `iele-gas.md` contains the semantics of the gas model of IELE. `welformdness.md` contains a formal definition of a well-formed IELE contract, a syntactically valid IELE contract free from type errors and other malformed instructions and/or functions. Finally, `data.md` contains the semantics of various data structures and utilities used in the rest of the specification. In the following paragraphs, we discuss examples from the formal specification of IELE along with features of K as needed.

Syntax The syntax of IELE is specified as a collection of EBNF-style productions. As an example, the productions shown in Listing 4 define the syntax for a IELE contract and its contents. The left-hand side of each syntax production defines a K sort and the right-hand side of the production gives one or more syntactically valid ways to construct a value of the sort. The keywords enclosed in double quotes represent terminal symbols. K uses these productions to automatically derive a parser for the language.

Execution State (Configuration) The execution state of a IELE program is defined as a K configuration, that is an ordered list of potentially nested cells, specified with an XML-like notation. At any given time during execution, each cell contains a corresponding value that reflects the current execution state. When declaring a K configuration, initial values for the cell contents are supplied. Among other components, the configuration for IELE contains a description of the local state and the network state.

1) The local state is created when a transaction is sent to a specific account and contains information about the smart contract code associated with the account, the intra-contract call stack, the amount of gas remaining, and the state of the local memory and virtual register file (see Listing 5).

The `id` cell contains the address of the currently executing account and the `caller` cell contains the address of the account that initiated the transaction. The `program` cell and its nested cells contain the code of the currently executing smart contract. The code is contained in one or more `function` cells, one for each function of the smart contract. These cells in turn contain the code for the corresponding function as a list of blocks and other information, such as the function name and number of formal parameters, and a jump table mapping label names to corresponding blocks. The `gas` cell contains the amount of gas remaining. This cell is initialized upon receiving the transaction to the amount

Listing 5. Configuration for local state

```
<id>     0 </id>     // Currently executing account
<caller> 0 </caller> // Account that called current account
<gas>    0 </gas>    // Current gas remaining
<program>
  <functions>
    <function multiplicity="*" type="Map">
      <funcId>    deposit </funcId>    // Name of the function
      <nparams>   0       </nparams>   // Number of parameters
      <jumpTable> .Map    </jumpTable> // Jump table
      <nregs>     0       </nregs>     // Number of registers
      <instructions>
        (.Instructions .LabeledBlocks):Blocks
      </instructions>                  // The blocks of the function
    </function>
  </functions>
  // ... more cells ...
</program>
<localCalls> .List  </localCalls> // Intra-contract call stack
<regs>       .Array </regs>       // Current values of registers
<localMem>   .Map   </localMem>   // Current values of local memory
```

of gas sent from the caller and is being reduced while the smart contract code executes. Finally, the cells `regs` and `localMem` map virtual register names and local memory addresses to their containing values, while the cell `localCalls` contains a stack of frames for all the functions in the current call stack.

2) The network state contains information about the Ethereum network, such as active accounts, their balance in Ether, their storage contents, whether or not they are associated with code, pending transactions, and more. The Ethereum network state at any point in time can be reached by replaying all the transactions that are stored in the blockchain up to this point. Instead of specifying the network state as a transaction log, we choose to describe only the current network state, as only it is relevant for the rest of the formal specification.

Listing 6 shows part of the network state that describes the state of active Ethereum accounts. The `accounts` cell contains one cell per account in the network. Active accounts have their address in the `acctID` cell and their balance in the `balance` cell. The `code` cell contains any associated smart contract code. The `storage` cell maps storage addresses to their current contents for the account's storage. The `nonce` cell contains a monotonically increasing integer that counts the number of transactions performed by this account.

Listing 6. Configuration for network state

```
<network>
  <accounts>
    <account multiplicity="*" type="Map">
      <acctID>   0          </acctID>     // ID of account
      <balance>  0          </balance>    // Funds in account
      <code>     #emptyCode </code>       // Contract of account
      <storage>  .Map       </storage>    // Permanent storage
      <nonce>    0          </nonce>      // Nonce of account
    </account>
  </accounts>
  // ... more cells ...
</network>
```

Transition Rules Transition rules define valid rewrites of the current configuration to a next one. Each rule consists of a left-hand side that is a pattern over one or more configurations (meaning it may contain variables) and a right-

hand side that describes how a matched configuration should be rewritten to the next valid configuration. A pattern matches an actual configuration when there exists an assignment of its variables that makes it equal to the configuration. The derived interpreter that K generates matches patterns found in the left-hand side of rules with the contents of the current configuration, and rewrites it according to the right-hand side. The program verifier does the same, except that it applies the rules symbolically, using unification instead of matching.

The `k` cell of the configuration drives the execution: It contains at any time a list of execution steps to be matched and rewritten. The IELE semantics defines a set of internal operators that represent different such execution steps and maintains a list of such operators inside the `k` cell during execution.

As an example, Listing 7 shows the rules that specify the behavior of the `div` instruction. The syntax production defines the internal operator `#exec`, which represents the execution of a single IELE instruction. Both rules match a configuration where the top of the `k` cell contains the `#exec` operator with a division instruction. The first rule matches when the denominator is different from zero, as specified in the `requires` clause. Then, the top of the `k` cell is rewritten to another internal operator, `#load`, that loads the result of the division to the left-hand side virtual register. The `/Int` and `=/=Int` operators are K built-in operators for arbitrary-precision signed integers.

Listing 7. Rules for the `div` instruction

```
syntax InternalOp ::= "#exec" Instruction
// ------------------------------------------------
rule <k> #exec REG = div W0 , W1 => #load REG W0 /Int W1  ... </k>
     requires W1 =/=Int 0
rule <k> #exec REG = div W0 ,  0 => #exception USER_ERROR ... </k>
```

The second rule matches in the case of division by zero and rewrites the top of the `k` cell to an `#exception` with the appropriate error code (here `USER_ERROR` is a macro that stands for corresponding error code). Other parts of the specification provide rules that handle exceptions by reverting all account state changes since the account started execution for the current transaction and returning the error code to the caller. The ellipses (`...`) is K syntax for a pattern that matches the rest of the `k` cell, which is a list of internal operators.

Listing 8. Rules for the `#load` internal operator

```
syntax InternalOp ::= "#load" LValue Int
                    | "#load" Int Int Int [klabel(#loadAux)]
// ---------------------------------------------------------
rule <k> #load % REG VALUE => #load REG VALUE {REGS [ REG ]} ...</k>
     <regs> REGS </regs>
rule <k> #load REG VALUE OLD => . ... </k>
     <regs> REGS => REGS [ REG <- VALUE ] </regs>
     <currentMemory>
       CURR => CURR -Int intSize(OLD) +Int intSize(VALUE)
     </currentMemory>
```

The rules shown in Listing 8 specify the behavior of the `#load` internal operator, used to store values in virtual registers. Note that the syntax for a virtual register (after desugaring) is `% Int` and the integer that is the name of the register is used as an index in the register file to look up its value.

The first rule accesses the current register file in the `regs` cell and looks up the old value of the register to be updated. It then rewrites the `#load` operator to an auxiliary operator that matches the second rule. The second rule updates the register file using the K built-in operator `_[_<-_]` for writing array elements. It also updates the total size of the register file in the `currentMemory` cell; this information is needed to compute the gas cost of the operation. The top of the `k` cell is rewritten to ".", which is a K idiom for the empty string.

As a last example, `intSize`, used above to compute the size in 64-bit words of the given arbitrary-precision signed integer, is defined as shown in Listing 9.

Listing 9. Rules for the `intSize` operator

```
syntax Int ::= intSize ( Int ) [function]
// ----------------------------------------
rule intSize(N) => (log2Int(N) +Int 2) up/Int 64 requires N >Int 0
rule intSize(0) => 1
rule intSize(N) => intSize(~Int N) requires N <Int 0
```

The syntax production specifies the pattern `intSize(_)` as a member of the `Int` sort (the arbitrary-precision signed integers) and attaches the `function` attribute to it. This attribute informs K that the pattern is "pure", as in the rules for rewriting it do not depend on any context other than its argument. The K rewrite engine will attempt to rewrite these pattern as much as possible when they appear anywhere in the current configuration.

5 Formal Verifier of IELE Smart Contracts

K provides a sound and relatively complete language-parametric program verifier. That is, given a language semantics as input, K yields a program verifier for that language that can prove, modulo a domain reasoning oracle (currently Z3 [9]), any reachability property about any program in that language. This important capability of K, that generalizes and eliminates the need for Hoare logic, has already been demonstrated with languages that are much larger than IELE, such as C, Java and JavaScript, and shown to offer the same level of automation and performance as program verifiers crafted specifically for the languages in question (e.g., VCC for C) [7]. Here we briefly explain how the K verifier works with IELE as input language, to verify IELE smart contracts. We emphasize how it compares with the same generic verifier instantiated with the EVM semantics [19], which has been used extensively as part of commercial services [31,37].

As discussed in Section 3, formal verification of smart contracts was one of the main forces driving the design of IELE, often in sharp contrast to the design of the EVM: statically known jumps allow us to write and compose code properties modularly; eliminating stack bounds for arithmetic expression evaluation allows us to soundly focus only on functional properties of code and write simpler, higher-level properties; eliminating the ABI encoding conventions allow us to reason about any programs, not only about those that obey the conventions; unbounded integers eliminate the need to reason about arithmetic overflow; etc.

We now discuss how to verify in IELE the most popular smart contract, ERC20 [43], which provides functionality for maintaining and exchanging tokens. Details about the verification of ERC20 in EVM can be found in [31,37].

Formal Specification We start with ERC20-K [35], a complete language-independent formalization of the high level business logic of the ERC20 standard. ERC20-K clarifies what data (e.g., balances and allowances) are handled by the various ERC20 functions and the precise meaning of those functions on such data. More importantly, ERC20-K clarifies the meaning of all the corner cases that the ERC20 standard omits to discuss, such as transfers to itself and transfers that result in arithmetic overflow. From ERC20-K, we can easily derive the specification for the ERC20 contract written in IELE by mapping ERC20-K to the configuration of IELE. We follow the same approach and DSL used for EVM [31], but taking the specifics of IELE into account. Mathematically, the ERC20 specification consists of a set of reachability formulae of the form $\phi \Rightarrow \psi$, with the meaning that the set of states satisfying/matching the pattern ϕ will either reach a state in ψ or not terminate when executed with IELE.

Listing 10 shows a snippet of the specification for two possible behaviors of `transfer`. For each case, it specifies the function name (`fid`), the function parameters (`callData` and `regs`), the return value (`output`), whether an exception occurred (`k`), the log generated (`log`), the storage update (`storage`), and the path-condition (`requires`). The success case (`[transfer-success]`) specifies that the function succeeds in transferring the `VALUE` tokens from the `FROM` account to the `TO` account, with generating the corresponding log message, and returns 1 (i.e., true), provided that the `FROM` account has sufficient balance. The failure case (`[transfer-failure]`) specifies that the function throws an exception without modifying the account balances, if the `FROM` balance is insufficient.

Listing 10. Formal specification of ERC20 `transfer`

```
[transfer-success]
k: #execute => #end ...
callData: TO, VALUE, .Ints
output: _ => (1, .Ints)
regs: (0 |-> TO_ID  1 |-> VALUE  .Map) => _
fid: transfer
log: ... (. => #eventLog(FROM, #topics(TransferEvent, FROM, TO), Int2Bytes(VALUE)))
storage: #mapKey({BALANCE}, FROM) |-> (BAL_FROM => BAL_FROM -Int VALUE)
         #mapKey({BALANCE}, TO)   |-> (BAL_TO   => BAL_TO   +Int VALUE)
         ...
requires: andBool 0 <=Int VALUE
          andBool FROM =/=Int TO
          andBool VALUE <=Int BAL_FROM

[transfer-failure]
k: #execute => #exception(4) ...
callData: TO, VALUE, .Ints
output: _ => _
regs: (0 |-> TO_ID  1 |-> VALUE  .Map) => _
fid: transfer
log: ...
storage: #mapKey({BALANCE}, FROM) |-> BAL_FROM
         #mapKey({BALANCE}, TO)   |-> BAL_TO
         ...
requires: andBool 0 <=Int VALUE
          andBool FROM =/=Int TO
          andBool VALUE >Int BAL_FROM
```

Formal Verification We verified the hand-written IELE implementation of ERC20 at [36], following the same automatic process used for EVM-level ver-

ification described in [31]. All 15 high-level ERC20 properties were seamlessly proved, automatically. It is insightful to compare the IELE and EVM verification experiences. We found that most of the EVM-level verification challenges described in [31] are addressed by the IELE language design. For example:

(Arithmetic Overflow) Since EVM performs modular arithmetic (i.e., mod 2^{256}), detecting arithmetic overflow is critical for preventing security attacks. When writing EVM-level specifications, one needs to reverse engineer constraints on the input such that the arithmetic overflow checks in the program pass. For example, in case `transfer-success`, one needs to add constraints to make sure that the balance of `TO` account does not overflow. This is not only tedious, but also imposes non-trivial proof obligations on SMT solvers. In contrast, IELE arithmetic instructions admit unbounded integers, which not only makes the smart contract more secure but also makes it much easier to specify correctly.

(Hash Collision) Due to the storage limitation of EVM, compilers such as Solidity and Vyper use SHA3 hash to implement the builtin map. However, SHA3 is not cryptographically collision free. The contract developers simply assume collisions will not occur during normal execution and SHA3 hash is modeled as an injective function during the verification of EVM smart contracts. Unfortunately, that is unsound: one can derive false, using the pigeonhole principle. In contrast, IELE provides *infinite* memory and storage, so injective functions can be defined to map different keys to different locations instead of SHA3 hashing.

6 Conclusion

We presented IELE, a new low-level language for smart contracts. The full formal specification of IELE is available as a K specification, serving at the same time as the implementation as well as a formal documentation of the language. The specification/implementation of IELE is in par with the state of the art in the world of smart contract virtual machines. With the support of a Solidity to IELE compiler (see appendix B.1) and a fully functional Ethereum client that is based on Mantis and powered by the IELE virtual machine (see appendix B.2), we were able to deploy and execute IELE contracts in a real-world blockchain testnet. This makes IELE the first practical language to be defined and implemented directly as a formal semantics specification and significantly raises the bar for how virtual machines for the blockchain must be developed. In this new and disruptive field where security and correctness are paramount, it is in our view unjustified to adopt any lower standards anymore.

Acknowledgements

We are grateful to IOHK (http://iohk.io) for funding the IELE project, as well as for insightful discussions, encouragements and constructive criticisms along the way. The work on the K framework and its tooling was supported in part by NSF grant CNS 16-19275 and by the United States Air Force and DARPA under Contract No. FA8750-18-C-0092.

A Appendix: IELE Gas Model

A gas model assigns a cost in "gas" to each instruction executed, and transaction senders specify the maximum amount of gas to be used to run their transaction. The assigned gas costs are fixed by the language definition (but transaction senders choose the price they offer per unit of gas consumed).

The gas costs are meant to be proportional to computation time, with additional charges for the amount of memory used during contract execution, and larger costs for operations changing persistent state. In addition to all the factors that need to be considered when designing a gas model for the EVM, in IELE we also have to consider the fact that integers are unbounded and thus arithmetic operations on them can become arbitrarily complex. In particular, we should chose the size-dependence of operations so there is no incentive to explicitly implement arbitrary-precision arithmetic algorithms in IELE instead of using the provided instructions.

Our gas costs for computation time are based on measuring relative performance of instructions on various input sizes. For arithmetic operations significant time is spent in the GMP library [17] which K uses to implement arbitrary-precision arithmetic, and we checked the asymptotic performance of the algorithms which GMP uses for the relevant input sizes. Many operations have performance linear in the input, and have a simple cost formula with a constant cost and a linear factor in input size. An example of a linear computational cost formula is the formula for bitwise `and`, while an example of a constant cost formula is the formula for `iszero`, an operation testing whether a number is zero. Both formulas are shown in Listing 11.

Listing 11. Gas formulas for bitwise `and` and `iszero`

```
rule #compute [ _ = and W0, W1 ]
  => Gbitwise + min(size(W0),size(W1)) * Gbitwiseword

rule #compute [ _ = iszero W ] => Giszero
```

More complicated formulas were needed for multiplication, division, modulus, and modular exponentiation. For some of these operations inputs up to several kilobits are commonly used in cryptography, so we considered inputs up to several kilobytes. We found a simple quadratic cost model was not sufficient across this range, but taking into account the Karatsuba algorithm [24] was sufficient (on larger inputs more asymptotically efficient algorithms will dominate, so a Karatsuba-based cost formula remains an overestimate).

Karatsuba multiplication has a complexity dominated by $n^{log_3 2}$. The definition in Listing 12 approximates that value for a given n with a family of quadratic functions of the form $a2 * n^2 + a1 * n + a0$, such that the approximation is continuous and differentiable and its derivative continuous.

Listing 12. Approximating $n^{log_3 2}$

```
rule #overApproxKara(N) =>
  #if       N <=Int   32 #then N *Int N
  #else #if N <=Int 1024
        #then N *Int N /Int  4 +Int  48 *Int N -Int    768
        #else N *Int N /Int 16 +Int 432 *Int N -Int 197376
  #fi #fi
```

Argument sizes are counted in 64-bit words, which corresponds to the actual threshold where costs change on 64-bit hardware, and may also allow simplified gas estimation that ignores input sizes to be useful for some code. Exponentiation without modulus is the major exception, because of the drastic dependence of result size on the value of the exponent the cost calculation considers the exact position of the highest bit set in the highest word of the exponent (computed using the `#adjustedBitLength` function):

Listing 13. Approximating gas costs for exponentiation

```
rule #compute [ _ = exp W0 , W1 ] => Cexp(intSize(W0), W0, W1)
rule Cexp(L1, W1, W2)
  => Gexpkara *Int #overApproxKara(#adjustedBitLength(L1, W1)
              *Int W2 /Int 64)
     +Int Gexpword *Int L1
     +Int Gexp
```

The gas cost for memory is based on peak memory consumption. Memory consumption includes values in registers and in local memory, and also stack frames for contract-local function calls. We based our costs on the design of EVM, where memory consumption has a quadratic cost. EVM is a stack based language and has a hard limit on stack size rather than accounting for stack size as part of their gas cost for memory, so we allow an amount of memory up to the maximum size of the EVM stack to be used with no charge. Whenever peak memory consumption increases the incremental cost is charged immediately. While memory usage remains below the previous peak there is no charge for increasing or decreasing the amount of memory used. To account for operations where the result size can not be precisely predicted from the input size alone, some operations use an upper bound of the result size instead: the gas is charged according to the the effect of the upper bound but the peak is actually updated after the operation and according to the actual result size.

Memory costs for computational operations that only return results in a register are defined in terms of an auxiliary function `#registerDelta` which updates peak memory and assesses gas cost given the target register and the upper bound on result size; see Listing 14.

Listing 14. Updating peak memory and assesing gas costs

```
rule <k> #registerDelta(% REG, NEWSIZE) => #deductMemory(PEAK)...</k>
     <currentMemory> CURR </currentMemory>
     <regs> REGS </regs>
     <peakMemory>
       PEAK => maxInt(PEAK,
                      CURR +Int NEWSIZE -Int intSize({REGS[REG]}))
     </peakMemory>
```

Most bitwise operations return results at most as large as an input. Addition or subtraction can produce a result one word larger than an input. Similarly to the gas costs rules, the result size for exponentiation without modulus considers the exact position of the highest bit set in the highest word of the exponent. For example, the formulas for `or`, `add`, and `exp` are shown in Listing 15.

For storage based costs, operations which transfer cryptocurrency, create new accounts, or modify persistent storage have high costs, reflecting that these per-

Listing 15. Gas formulas for `or`, `add`, and `exp`

```
rule #memory [ REG = or W0, W1 ]
  => #registerDelta(REG, max(size(W0),size(W1)))

rule #memory [ REG = add W0 , W1 ]
  => #registerDelta(REG, max(size(W0), size(W1)) +1)

rule #memory [ REG = exp W0 , W1 ]
    => #registerDelta(REG, #adjustedBitLength(intSize(W0), W0)
       *Int W1 /Int 64)
```

manently increase the size of blockchain history. Operations which query persistent state also have elevated costs, since they might require retrieval of data from secondary storage. Here a major departure from EVM is that we do not charge for programmatically creating a new smart contract account from contract code that has already been uploaded to the blockchain. This is a safe replacement for cases where EVM programs would use the dangerous `delegatecall` instruction purely to reduce the cost of creating smart contracts sharing behavior. For example, the cost `Gbalance` of looking up the account balance is greater than 100 times the cost `Gbitwise` of bitwise operations.

Overall, our gas model is designed to allow for instructions to work on arbitrarily large values, and to allow IELE to run without artificial limits on the size of data or call stacks, while preserving the existing goals of the EVM gas model.

B Appendix: IELE Infrastructure and Evaluation

B.1 ISOLC: The Solidity to IELE compiler

Solidity [12] is the most popular high-level language for Ethereum smart contracts, and has been used to implement tens of thousands of contracts [16], some of which handling hundreds of millions USD [28]. Because of Solidity's popularity and in order to have an automated way to obtain IELE contracts for testing, we developed a Solidity-to-IELE compiler, *isolc* [39]. This was a necessary, albeit manual and tedious, 9 man-month endeavor. Our compiler reuses much of the front-end of the Solidity-to-EVM compiler, *solc*, but implements a new back-end to generate IELE code. We used *isolc* to compile about 1000 Solidity smart contracts of the *solc* test suite to IELE, including real-world contracts, such as a multi-signature wallets and ERC20 compliant tokens.

At the time of writing, the compiler supports most of the latest version of Solidity (version 0.4.23) with some additional features. The Solidity features that are not supported are those exposing low-level EVM features that IELE deliberately does not support, as discussed in Section 3. The additional features are taking advantage of the new IELE features and serve as replacements for the unsupported features. Below we discuss representative examples.

Low-level calls Solidity supports a low-level account call built-in that has similar semantics with an EVM call: The caller can use it to send an arbitrary bytestream of data to the callee. For security, IELE disallows such low-level operations; normal calls can and should be used instead. Hence we drop support for this built-in when compiling to IELE.

delegatecall is another low-level Solidity built-in for account calls that exposes the corresponding EVM functionality. As discussed in Section 3.2, for security, IELE dropped this functionality (and the low-level calls); see also Section B.3. Hence our Solidity compiler does not support this built-in.

Arbitrary-precision integer types Solidity only provides fixed-width integer types with bit width that can vary from 1 to 256 bits. We add two new arbitrary-precision integer types (a signed and an unsigned type) to take advantage of IELE's arbitrary-precision arithmetic.

The full list of feature differences can be found in the *isolc* repository [39]. We also provide examples of how to avoid/replace these features in existing Solidity smart contracts, so that they can be compiled to IELE.

B.2 IELE VM and Integration with Blockchain

Ethereum transactions modify the global state of the network by transferring Ether between accounts and optionally invoking and/or deploying smart contract code in the blockchain. These transactions are also stored in the blockchain and can be replayed by an Ethereum client to get a current view of the network's global state. In a IELE-based network, when IELE code is invoked and/or deployed as part of a transaction, a IELE VM executes the code and updates the network state. We have implemented such a VM using two tools both auto-generated from the language specification: a IELE interpreter and a IELE well-formedness checking tool.

We have also integrated our IELE VM with the Mantis Ethereum client [22]. Using Mantis we have successfully deployed and executed IELE smart contracts on an Ethereum-based blockchain [20]. While replaying transactions, Mantis sends relevant transactions to the IELE VM. In turn, the IELE VM parses incoming transactions to figure out which function of which account needs to be invoked and then executes the corresponding IELE code using the interpreter. In case the transaction involves creation of a new account and deployment of smart contract code, the IELE VM uses the well-formedness checker to verify the code being stored in the blockchain. During IELE code execution, Mantis provides the VM with information about the current global state (e.g. contents of an account's storage). Finally, the IELE VM uses the output of the interpreter to determine the modifications on the global network state caused by the transaction and communicates those to Mantis.

B.3 Evaluation

In this section we evaluate the following claims:

- The formal specification of IELE is adequate. The automatically generated IELE VM is then also adequate.
- The Solidity to IELE compiler is also adequate.
- The design approach based on formal semantics is realistic in terms of effort.
- The IELE VM has acceptable performance.
- The IELE design prevents exploits found in EVM contracts that have cost millions of USD in the past.

Adequacy To demonstrate that the IELE language specification, the derived IELE VM, and the Solidity to IELE compiler are adequate, we test all these tools in a pipeline similar to the one used by real-world smart contract developers: We compile a suite of Solidity contracts to IELE and then deploy the IELE contracts to an Ethereum blockchain and execute them using the IELE VM and the Mantis client. We use the test suite of the Solidity to EVM compiler [13], which is also used to test the production version of the Solidity to EVM compiler. We assume that a test case has passed when the following are true: the contract compiles to IELE successfully, the IELE code is deployed successfully, and the expected result is returned by the deployed code when it is invoked with the test's input. Our pipeline succeeds in 100% of the tests, thus providing high confidence on the adequacy of both the Solidity to IELE compiler and the IELE specification/implementation.

Design and Implementation Effort Although the formal specification of IELE is larger than that of the EVM, the design, formal specification, implementation, and testing of IELE took an estimated 10 man-months. Various lower level data structures and functions are used across the semantics definition, and we tried to reuse the publicly available KEVM definition [19] as much as possible for this part of the specification. We argue that this is a short amount of time for designing a full language along with its formal specification and a reference implementation.

Performance evaluation We evaluate two aspects of the IELE implementation: the binary size generated from the Solidity to IELE compiler and execution speed of IELE VM. For baseline, we use the Solidity to EVM compiler (version 0.4.21, the most recent version of the compiler compatible with KEVM) and the KEVM.

Binary Size We compiled the Solidity unit tests and 5 large contracts to both EVM and IELE and measured the size of the resulting binary IELE and EVM programs. The geometric mean of the ratios of the EVM binary size over the corresponding IELE binary size for each of the unit tests is 1.18, meaning that the IELE binary is on average about 84% of the size of the corresponding EVM binary. For the large contracts, the ratio is 0.85, meaning that the IELE binary is on average about 117% of the size of the corresponding EVM binary. The size reduction in the small programs is due to the fact that IELE is a register-based language and thus avoids stack manipulation opcodes like PUSH found in EVM programs. However our current compiler uses a naive code generation algorithm, with no optimizations, and as a result generates more code for larger programs. The IELE binary size will be reduced significantly as the compiler matures.

Execution Time We deployed and executed the above programs using Mantis along with the IELE VM and KEVM. The experiment ran on an AMD Ryzen Threadripper 1950X @1.9GHz processor with 16 cores and 64GB of memory. All the processing is single-threaded. The geometric mean of the ratios of the EVM transaction time over the corresponding IELE transaction time for the unit tests

is 1.38, meaning that the IELE VM is on average 1.38 times faster than KEVM. For the large contracts, the ratio is 1.18.

These results show that the Solidity to IELE compiler and the IELE VM have acceptable performance. These improvements in performance over KEVM should be regarded as a bonus for a cleaner design and the choice of a register-based architecture, because as explained in Section 3, the design goals of IELE were not related to performance.

Security Here we review three costly recent exploits of smart contracts. We discuss the exploit and how it could be avoided in a IELE-based blockchain.

Parity Wallet Freeze (Nov 2017) [41] Parity's wallet smart contract can be associated with accounts of Ethereum users to provide them with more advanced functionality than simply transferring Ether, such as transaction logging, setting withdrawal limits, etc. In order to avoid code duplication, the wallet smart contract does not contain any code that mutates the user account's state. All the state-modifying logic is deployed as a library smart contract and associated with a single account. Wallet accounts invoke code from this library using EVM's `delegatecall`, a special account call where the called library code modifies the state of the caller account instead of its own. Unfortunately, a bug in the library smart contract allowed an attacker to take ownership of the account and subsequently delete it. As a result, the wallet accounts lost any ability to modify their state, including transferring their balances elsewhere. The cost of this attack has been estimated to about 150M USD of funds frozen in over 500 wallet accounts. Note that, by the nature of the blockchain, accepted transactions are irreversible, so it is not possible for Parity to roll back the deletion transaction or for users to roll back the wallet code deployment transactions.

Listing 16. Solidity contract that showcases the DAO exploit

```
contract Vulnerable {

   mapping(address => uint) userBalances;

   function addToBalance() payable {
     userBalances[msg.sender] = userBalances[msg.sender] + msg.value;
   }

   function withdrawBalance() {
     uint amountToWithdraw = userBalances[msg.sender];
     // The next line performs an account call back to
     // the caller of this function, sending Ether equal
     // to the value of ammountToWithdraw.
     // The low-level Solidity call is used with no arguments,
     // meaning that the fallback function of the receiver
     // will be invoked.
     msg.sender.call.value(amountToWithdraw)();

     userBalances[msg.sender] = 0;
   }
}
```

This kind of exploit reveals one of the many dangers of the `delegatecall` functionality of EVM. IELE has a completely different model for code reuse that does not rely on libraries and *delegatecall* (see Section 3.1). In fact `delegatecall` is not part IELE, hence making such an attack impossible.

Parity Wallet Breach (Jul 2017) [4] This attack is again related to Parity's wallet. The wallet smart contract has support for multiple owner accounts that have access to the wallet account funds. The wallet account keeps a list of the owner accounts in its storage. This list is initialized by a separate transaction after the wallet contract is deployed, since EVM does not allow storage initialization at contract creation time. Instead, the wallet smart contract includes an owner initialization function that has to be called at a later time. The attack took advantage of a bug in the wallet's design, where the owner initialization function could be called by anyone. The attackers reinitialized ownership of various wallet accounts to themselves giving them the ability to withdraw those accounts' balances. More than 30M USD was stolen, before wallet users became aware of the attack and transferred their funds elsewhere.

The IELE design does not necessarily prevent someone from writing a contract that allows anyone to reinitialize its state, but IELE supports storage initialization at contract creation time through the `@init` function (see Section 3.1). In a IELE-based blockchain, it is not necessary to create contracts and then initialize them, as is the case for EVM. Contract deployment and initialization in IELE can be done in a single step: `@init` is called at contract creation time and can contain initialization code. Moreover, `@init` is private, so it cannot be invoked by other accounts through a call.

The DAO attack (Jun 2016) [8] A simplified version of the vulnerable contract in the DAO attack is shown in Listing 16. The vulnerable contract functions as a bank, where client accounts deposit Ether to the contract's balance and the contract maintains a mapping, `userBalances`, from each client account to its balance. The function `addToBalance` accepts an Ether amount (accessible within the function as `msg.value` from the caller account (accessible within the function as `msg.sender`) and records the deposited amount in the contract's mapping. The vulnerable function named `withdrawBalance` sends back to the caller account the whole amount of Ether that this account has so far deposited. The function looks up the current total balance of the caller account in the contract's mapping, sends that amount of Ether to the caller account, and finally updates the caller account's balance to 0. Since sending Ether involves performing a call back to the caller account, `withdrawBalance` uses the low-level `call` of Solidity (Section B.1). This callback (where no particular function is called) will invoke the so-called fallback function of the client, and a malicious client can invoke `withdrawBalance` again from its fallback function. Thus `withdrawBalance` is reentered after the Ether has been sent but before the balance is updated in the contract's map.

The attacker exploited this vulnerability and stole more than 50M USD from the contract. This attack prompted for the first time a hard fork in the Ethereum blockchain [6].

Of course, the exploit is eliminated by reordering the sending of Ether and the map update. However, there is a more disciplined way to eliminate it: using the high-level Solidity built-ins `transfer` or `send` instead of using the low-level `call`. These built-ins provide the callee with a pre-defined very low amount of

gas that is not even enough for a single callback from the callee. On the other hand the low-level `call` provides the callee with a custom amount of gas, by default the whole remaining gas of the caller. As discussed in Section B.1, the Solidity to IELE compiler does not support the low-level `call`, hence making this particular case of re-entrancy inexpressible.

References

1. Atzei, N., Bartoletti, M., Cimoli, T.: A survey of attacks on Ethereum smart contracts. IACR Cryptology ePrint Archive **2016**, 1007 (2016), https://eprint.iacr.org/2016/1007.pdf
2. Blockstream: Simplicity blog post and resources (2019), https://blockstream.com/2018/11/28/en-simplicity-github/
3. Bogdanas, D., Rosu, G.: K-Java: A Complete Semantics of Java. In: Proceedings of the 42nd Symposium on Principles of Programming Languages (POPL'15). pp. 445–456. ACM (January 2015). https://doi.org/10.1145/2676726.2676982
4. Breidenbach, L., Daian, P., Juels, A., Sirer, E.G.: An in-depth look at the parity multisig bug (2017), http://hackingdistributed.com/2017/07/22/deep-dive-parity-bug/
5. Buterin, V., Ethereum Foundation: Ethereum White Paper. https://github.com/ethereum/wiki/wiki/White-Paper (2013)
6. del Castillo, M.: Ethereum executes blockchain hard fork to return DAO funds (2016), http://www.coindesk.com/ethereum-executes-blockchain-hard-fork-return-dao-investor-funds/
7. Ştefănescu, A., Park, D., Yuwen, S., Li, Y., Roşu, G.: Semantics-based program verifiers for all languages. In: Proceedings of the 31th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'16). pp. 74–91. ACM (Nov 2016). https://doi.org/http://dx.doi.org/10.1145/2983990.2984027
8. Daian, P.: DAO attack (2016), http://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/
9. De Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: TACAS'08. LNCS, vol. 4963, pp. 337–340 (2008)
10. Ellison, C., Rosu, G.: An executable formal semantics of C with applications. In: Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'12). pp. 533–544. ACM (January 2012). https://doi.org/10.1145/2103656.2103719
11. Ethereum: Ethereum C++ Client (2019), https://github.com/ethereum/cpp-ethereum
12. Ethereum: Solidity documentation (2019), http://solidity.readthedocs.io
13. Ethereum: Solidity github project (2019), https://github.com/ethereum/solidity
14. Ethereum: Vyper documentation (2019), https://vyper.readthedocs.io
15. Etherscan: Ethereum Transaction Growth. https://etherscan.io/chart/tx (2019)
16. Etherscan: Ethereum Verified Contract Sources. https://etherscan.io/contractsVerified (2019)
17. Free Software Foundation, I.: Gnu mp documentation (2019), https://gmplib.org/manual
18. Hathhorn, C., Ellison, C., Rosu, G.: Defining the undefinedness of C. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15). pp. 336–345. ACM (June 2015). https://doi.org/10.1145/2813885.2737979

19. Hildenbrandt, E., Saxena, M., Zhu, X., Rodrigues, N., Daian, P., Guth, D., Moore, B., Zhang, Y., Park, D., Ştefănescu, A., Roşu, G.: KEVM: A complete semantics of the Ethereum virtual machine. In: 2018 IEEE 31st Computer Security Foundations Symposium. pp. 204–217. IEEE (2018). https://doi.org/10.1109/CSF.2018.00022
20. IOHK: IELE Testnet. https://testnet.iohkdev.io/iele/ (2019)
21. IOHK: KEVM Testnet. https://testnet.iohkdev.io/kevm/ (2019)
22. IOHK: Mantis Ethereum Classic Client (2019), https://iohk.io/blog/mantis-ethereum-classic-beta-release
23. IOHK: Plutus testnet (2019), https://testnet.iohkdev.io/plutus/
24. Karatsuba, A., Ofman, Y.: Multiplication of many-digital numbers by automatic computer. In: Dokl. Akad. Nauk SSSR. vol. 145, pp. 293–294 (1962), http://mi.mathnet.ru/eng/dan26729
25. KEVM: Jello paper (2019), https://jellopaper.org/
26. Lattner, C., Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization. pp. 75–. CGO '04, IEEE Computer Society, Washington, DC, USA (2004), http://llvm.org
27. Milner, R., Tofte, M., Harper, R., MacQueen, D.: The definition of standard ML : revised. MIT Press, Cambridge, Mass (1997)
28. Multisig: Multi-signature Wallet Ethereum Account. https://etherscan.io/address/0xab7c74abc0c4d48d1bdad5dcb26153fc8780f83e#code (2019)
29. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system (2008), https://bitcoin.org/bitcoin.pdf
30. Park, D., Stefanescu, A., Rosu, G.: KJS: A complete formal semantics of JavaScript. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15). pp. 346–356. ACM (June 2015). https://doi.org/10.1145/2737924.2737991
31. Park, D., Zhang, Y., Saxena, M., Daian, P., Roşu, G.: A Formal Verification Tool for Ethereum VM Bytecode. In: Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'18). ACM (November 2018). https://doi.org/10.1145/3236024.3264591
32. PeckShield: New batchOverflow Bug in Multiple ERC20 Smart Contracts (CVE-2018–10299) (2018), https://medium.com/coinmonks/alert-new-batchoverflow-bug-in-multiple-erc20-smart-contracts-cve-2018-10299-511067db6536
33. RChain Cooperative: Rchain and rholang documentation (2019), https://architecture-docs.readthedocs.io/
34. Rosu, G., Serbanuta, T.F.: An overview of the K semantic framework. Journal of Logic and Algebraic Programming **79**(6), 397–434 (2010), http://kframework.org
35. RuntimeVerification: ERC20-K: Formal Executable Specification of ERC20. https://github.com/runtimeverification/erc20-semantics (2017)
36. RuntimeVerification: ERC20 Token in IELE (2019), https://github.com/runtimeverification/iele-semantics/blob/master/iele-examples/erc20.iele
37. RuntimeVerification: Formal Smart Contract Verification. https://runtimeverification.com/smartcontract/ (2019)
38. RuntimeVerification: The formal semantics for IELE — source code (2019), https://github.com/runtimeverification/iele-semantics
39. RuntimeVerification: The Solidity to IELE compiler — source code (2019), https://github.com/runtimeverification/solidity

40. Solana, J.: $500K hack challenge backfires on blockchain lottery SmartBillions (2017), https://calvinayre.com/2017/10/13/bitcoin/500k-hack-challenge-backfires-blockchain-lottery-smartbillions/
41. Steiner, J.: Security is a process: A postmortem on the parity multi-sig library self-destruct (2017), http://goo.gl/LBh1vR
42. Tezos: Michelson documentation (2019), https://tezos.gitlab.io/master/index.html
43. The Ethereum Foundation: ERC20 token standard (2019), https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20-token-standard.md
44. Wood, G.: Ethereum: A secure decentralised generalised transaction ledger. Ethereum project yellow paper **151**, 1–32 (2014)
45. Zilliqa: Scilla language webpage (2019), https://scilla-lang.org/