IELE: An Intermediate-Level Blockchain Language Designed and Implemented Using Formal Semantics

THEODOROS KASAMPALIS, Runtime Verification, Inc. and University of Illinois at Urbana-Champaign
DWIGHT GUTH, Runtime Verification, Inc.
BRANDON MOORE, Runtime Verification, Inc.
TRAIAN ȘERBĂNUȚĂ, Runtime Verification, Inc.
VIRGIL ȘERBĂNUȚĂ, Runtime Verification, Inc.
DANIELE FILARETTI, Runtime Verification, Inc.
GRIGORE ROȘU, Runtime Verification, Inc. and University of Illinois at Urbana-Champaign
RALPH JOHNSON, Runtime Verification, Inc.

Most languages are given an informal semantics until they are implemented, so the formal semantics comes later. Consequently, there are usually inconsistencies among the informal semantics, the implementation, and the formal semantics. IELE is an LLVM-like language for the blockchain that was specified formally and its implementation, a virtual machine, generated from the formal specification. Moreover, its design was based on problems observed formalizing the semantics of the Ethereum Virtual Machine (EVM) and from formally specifying and verifying EVM programs (also called "smart contracts"), so even the design decisions made for IELE are based on formal specifications. A compiler from Solidity, the predominant high-level language for smart contracts, to IELE has also been implemented, so Ethereum contracts can now also be executed on IELE. The virtual machine automatically generated from the semantics of IELE is shown to be competitive in terms of performance with the state of the art and hence can stand as the de facto implementation of the language in a production setting. Indeed, IOHK, a major blockchain company, is currently experimenting with the IELE VM in order to deploy it as its computational layer in a few months. This makes IELE the first practical language that is designed and implemented as a formal specification. It took only 10 man-months to develop IELE, which demonstrates that the programming language semantics field has reached a level of maturity that makes it appealing over the traditional, adhoc approach *even for pragmatic reasons.*

1 INTRODUCTION

A smart contract [Szabo 1994] is a program that implements the terms of a contract. Recently, smart contracts have been widely used to facilitate digital transactions in blockchain networks. Ethereum [Buterin and Ethereum Foundation 2013], with more that 500,000 daily transactions [Etherscan 2018a] and a market capacity reaching 50 billion USD [CoinMarketCap 2018], is the largest blockchain network supporting smart contract transactions. The smart contracts used in the context of Ethereum transactions are written in the language of the Ethereum Virtual Machine (EVM) [Wood 2014], 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 [Atzei et al. 2016; Breidenbach et al. 2017; Daian 2016; Solana 2017; Steiner 2017]. In response, KEVM [Hildenbrandt et al. 2018], the formal semantics of the EVM in K [Rosu and Serbanuta 2010], was developed to provide formal verification assistance to the EVM smart contract world, and has been welcomed by the Ethereum community. We were early adopters of KEVM, we

Authors' addresses: Theodoros Kasampalis, Runtime Verification, Inc. University of Illinois at Urbana-Champaign, theodoros.kasampalis@runtimeverification.com; Dwight Guth, Runtime Verification, Inc. dwight.guth@runtimeverification.com; Brandon Moore, Runtime Verification, Inc. brandon.moore@runtimeverification.com; Traian Șerbănuță, Runtime Verification, Inc. traian.serbanuta@runtimeverification.com; Virgil Șerbănuță, Runtime Verification, Inc. virgil.serbanuta@runtimeverification.com; Daniele Filaretti, Runtime Verification, Inc. daniele.filaretti@runtimeverification.com; Grigore Roșu, Runtime Verification, Inc. University of Illinois at Urbana-Champaign, grigore.rosu@runtimeverification.com; Ralph Johnson, Runtime Verification, Inc. ralph.johnson@runtimeverification.com.

carefully studied it and assimilated it in our projects and used it for formal verification of smart contracts. Through our own experience with KEVM and the reports of others, we became of aware of the limitations of EVM as a language design: the language includes several features that are easily exploitable and its low-level nature makes formal verification efforts tedious and time-consuming.

We have designed and implemented a new language for smart contracts named IELE that addresses EVM's limitations. IELE is an intermediate-level language with syntax similar to that of LLVM [Lattner and Adve 2004]. 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. Like EVM, IELE is the language of a corresponding virtual machine and an IELE contract will be executed on the virtual machine in the exact same form as it was verified.

IELE was designed using a formal specification, and its implementation was automatically generated from its specification using the same software used to generate the implementation of KEVM. 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 [Lattner and Adve 2004] and sometimes formal, like that of SML [Milner et al. 1997], but in either case the implementation is separate so it is not possible to execute test cases against the specification. Sometimes the language implementation comes first and standards follow. If there are many implementations, such as for C, it can be hard to capture the expected semantics of all the implementations.

To bridge the gap between specification and implementation, we use the K framework [Rosu and Serbanuta 2010] for both these tasks for IELE. The formal semantics of IELE is given as a K definition, a formal system of rewriting rules. The K Framework provides support for developing such semantic definitions as well as a host of auto-generated tools for such a definition, such as a 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) for the language as its reference implementation. As a result, IELE is formally specified and its implementation is correct by construction and remains correct in the presence of additions/updates to the language with zero effort. To our knowledge, IELE is the first language designed this way: as a complete formal specification, and with a correct by construction reference implementation that is automatically derived from its specification.

IELE is an intermediate-level language suitable as a compilation target and we have implemented a compiler for the Solidity language [Ethereum 2018b], a popular high-level language for smart contracts that until now could only be compiled to EVM. We use the Solidity to IELE compiler to test IELE with real-world Solidity contracts. Our compiler is successfully tested with the full test suite of the Solidity to EVM compiler, which contains more than a 1000 smart contracts ranging from unit tests to real world contracts managing hundreds of millions USD, such as ERC20 [The Ethereum Foundation 2018a] and multisig [The Ethereum Foundation 2018b]. Additionally, we integrated our IELE VM with the Mantis Ethereum client [IOHK 2018b] to actually execute IELE contracts on an Ethereum blockchain. Both the Solidity to IELE compiler and the IELE VM are shown to be competive with the state of the art in terms of performance: the IELE binaries generated by the compiler are on average 17% bigger than EVM binaries and the IELE VM is on average 1.38 times faster than KEVM. Due to the extensive testing and exposure of IELE to the real world, we consider the IELE design mature and complete.

In collaboration with IOHK [IOHK 2018a], a major blockchain organization, we are currently deploying IELE to a testnet blockchain to make it available to the public and we plan to subsequently deploy IELE on a real blockchain.

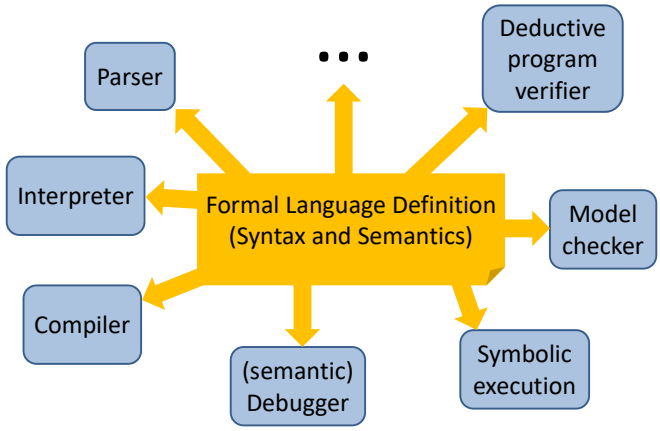

Fig. 1. The K vision [Rosu and Serbanuta 2010]

In summary, the contributions of this work are:

- We propose IELE, a new language for smart contracts designed from the ground up with the goals of security, formal verification, human-readability, and portability in mind.
- We present a new approach to language design, where the implementation is derived from the formal semantics specification, and is correct by construction. IELE is the first practical language designed and implemented this way. This approach is enabled by the K Framework.
- We develop useful tools for IELE: A compiler from the popular high-level smart contract language Solidity and a IELE virtual machine that integrates with the Mantis Ethereum client.

2 BACKGROUND

2.1 K Framework

The K Framework [Rosu and Serbanuta 2010] is a rewriting-based framework with foundations on Reachability Logic [Rosu et al. 2013] 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 methods 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. The verifier uses the same internal model for verifying programs, and that confidence carries over.

There exists a rich literature on using K for formalizing existing languages, such as C [Ellison and Rosu 2012; Hathhorn et al. 2015], Java [Bogdanas and Rosu 2015] and JavaScript [Park et al. 2015], among others. K has also been used to formally specify EVM [Hildenbrandt et al. 2018], 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 language specification [Wood 2014].

2.2 Blockchain and Ethereum

A blockchain is an append-only ledger that is commonly used for synchronization in distributed protocols. Cryptocurrencies such as Bitcoin [Nakamoto 2008], 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 between the accounts, and a set of so-called miners, computation resources that process pending transactions and append them in the blockchain. The blockchain itself is not kept in any central location, but rather stored locally by each client. A transaction is appended to the blockchain after being validated by the network clients through a consensus protocol. Each client can replay all the transactions in the blockchain to get a current view of the state of the network. Blockchain technology has wide financial applications [Nakamoto 2008; Narayanan et al. 2016; Sasson et al. 2014], but is also at the heart of a variety of open research problems, ranging from consensus protocols [Kogias et al. 2016] to decentralized storage systems [Wilkinson et al. 2014], and more [Ong et al. 2015].

Ethereum [Buterin and Ethereum Foundation 2013] is a blockchain-based network that aims to provide a decentralized, replicated computer for running distributed applications and uses a blockchain to store its global state. Ethereum also supports a programming language for writing *smart contracts*, which are programs associated with Ethereum accounts. Ethereum accounts interact with each other by the means of 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 may include, in addition to Ether, input data to be passed to the smart contract code. Similarly, the transaction receipt contains output data returned by the smart contract code execution.

Execution of smart contract code as part of a transaction should always terminate to ensure smooth operation and responsiveness of the Ethereum network. Since Ethereum is a decentralized system, there is no controller to ensure that transactions terminate. Ethereum solves this problem by introducing a resource, called *gas*, that is consumed by smart contract code execution: Each transaction is posted to the network along with a finite amount of gas, provided by the sender of the transaction, and if all available gas is consumed in smart contract code execution while resolving the transaction, the transaction is reverted. Within Ethereum, gas is purchased with Ether.

Ethereum is the largest blockchain network supporting smart contract transactions. Smart contracts on the Ethereum network 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.

2.3 EVM

Currently, Ethereum smart contracts should be written in the language of the Ethereum Virtual Machine (EVM) [Wood 2014]. The EVM is a stack-based virtual machine with an assembly-level language with no code/data separation or function-level calls/returns. The language is loosely specified in the Yellow Paper [Wood 2014], but has been recently formally specified as a K definition in KEVM [Hildenbrandt et al. 2018]. KEVM has provided a reference implementation that is considered for adoption by the Ethereum foundation, a deductive verifier that has been used to verify real-world contracts, and the Jello Paper [KEVM 2018], an english language specification of EVM generated from the documentation of the KEVM semantics definition.

The challenges uncovered by the KEVM effort on formal specification of the EVM language and verification of EVM programs, have been taken into account in our design of IELE to make IELE programs more secure and significantly easier to formally verify.

3 THE IELE LANGUAGE

The IELE language is designed to serve as an intermediate-level language for smart contracts. We identified five desired high-level properties for IELE and designed the language around them.

Security Smart contracts often manage large monetary amounts and have always been targets of attackers that seek to exploit any vulnerabilities in their code [Atzei et al. 2016]. Very often, language design weaknesses such as undefined behaviors, execution of arbitrary data as code, silent integer overflow act as enablers for attackers to exploit corresponding bugs in the code of a smart contract. 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 largest in terms of monetary amount exploits of Ethereum smart contracts are all due to software errors [Breidenbach et al. 2017; Daian 2016; Steiner 2017]. A strong defense against such exploits is formal correctness verification and IELE is designed with the goal of easy and practical formal verification in mind.

Human readable Smart contracts are intended to function as binding agreements between their 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 that they will be stored in the blockchain and executed. This is true for IELE contracts, since IELE syntax is very reminiscent of that of LLVM [Lattner and Adve 2004], a state-of-the-art intermediate language also designed to be human readable. Figure 2 compares the implementation of a simple sum function in IELE and EVM.

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. Consequently the derived implementation is also fully deterministic.

Gas model A necessary component of a language for smart contracts is a gas model that identifies elementary language constructs and assigns state-dependent gas costs to them according to their relative execution costs. 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 arbitrary-precision integer arithmetic, but the larger the numbers computed at runtime, the more gas required, and so on.

Listing 1 shows a simple forwarder contract in IELE. The contract forwards any amount of Ether [1] 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 if the

[1] Note that IELE can be used in any Ethereum-based blockchain where the cryptocurrency is called differently. We use Ether throughout the paper as it is the name of the Ethereum cryptocurrency.

forwarder 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 spent at the creator during the account call. Also note that if the forwarding fails (due to lack of gas or other reasons), 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 a K definition, is available in the IELE semantics GitHub repository, found here [Anonymous 2018a]. This formal semantics serves also as the reference implementation of the language. In the following subsections we give a high-level presentation of IELE, discuss the improvements over EVM, the state-of-the-art Ethereum language, and showcase our *formal semantics as implementation* approach to the design of the language.

3.1 IELE Contracts

A IELE contract is the main compilation unit of code that can be associated with an Ethereum account in the blockchain. A IELE contract has a name and contains one or more function definitions, global variable definitions, and external contract declarations. Public functions can be invoked by other Ethereum accounts while private functions are only callable from within the contract's code. Global variables are accessible from anywhere within the contract's code (but not from outside) and hold a constant value. Listing 1 shows a IELE contract with one global variable, `@creator`, one private function, `@init`, and one public function, `@deposit`.

Account Storage An Ethereum account includes a storage that is an unbounded sparse array of arbitrary-precision signed integers. The storage is persistent, meaning that it holds its contents throughout the account's lifetime. As such the storage contents of all Ethereum accounts are part of the global state and any modifications on them should be recorded in the blockchain.

```
  PUSH(1, 0);   PUSH(1, 0)
; MSTORE;       PUSH(1, 10);
; PUSH(1, 32); MSTORE
; JUMPDEST;     PUSH(1, 10)
; PUSH(1, 32); MLOAD; GT
; ISZERO;       PUSH(1, 43)
; JUMPI
; PUSH(1, 32); MLOAD
; PUSH(1, 0);   MLOAD; ADD
; PUSH(1, 0);   MSTORE
; PUSH(1, 1);
; PUSH(1, 32); MLOAD; SUB
; PUSH(1, 32); MSTORE
; PUSH(1, 10); JUMP
; JUMPDEST
; PUSH(1, 0);   MLOAD
; PUSH(1, 0);   SSTORE
```

(a) EVM

```
define public @sum(%n) {
  %result = 0

condition:
  %cond = cmp le %n, 0
  br %cond, after_loop
  %result = add %result, %n
  %n = sub %n, 1
  br condition

after_loop:
  ret %result
}
```

(b) IELE

Fig. 2. A simple function that computes the sum of the first n natural numbers in EVM and IELE

```
// This contract forwards any funds sent to it to the account that
// created it.
contract Forwarder {

  // global variable that holds the account storage index where we store 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 recieved 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()
  }
}
```

Listing 1. Forwarder contract in IELE

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 contract code can refer to those addresses by the name of the variable. See for example how the global variable `@creator` is used in the forwarder contract shown in Listing 1.

Contract Creation An Ethereum account can be created and associated with a IELE contract manually by an Ethereum end user, by posting of an appropriate transaction, or dynamically by another executing IELE contract, using IELE's `create` instruction. We say that the new account and/or the associated smart contract is being created by the creator account and/or its associated smart contract. We also say that the new contract is now deployed in the blockchain, 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 will dynamically create a new account, deploy the Forwarder smart contract and associate with it, and finally return the new account's address. The `send` attribute specifies that we do not want to send any initial amount of Ether to the new account.

```
// This contract implements a wallet that creates new accounts that forward funds
// sent to them to this wallet.
contract Wallet {
  external contract Forwarder

  define @init() {}

  // @deposit is empty meaning that 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()
  }
}
```

Listing 2. Wallet contract that creates Forwarder contracts

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. This means that 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, we see that the `Wallet` contract declares the `Forwarder` contract as external. This means that if a new `Wallet` contract is to be created and deployed in the blockchain, 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.

As we see, dynamic contract creation is guaranteed to only use code that has already become available in the blockchain, and no dynamic code generation is allowed. This design has two major advantages. Expensive code validation checks (e.g. well-formedness checks, formal verification checks etc.) need only take place when account creation transactions are posted and never during smart contract 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 so that no duplicate contract code is stored and accounts need only to store a pointer to their associated code. This allows for cheap dynamic account creation (only a pointer copy) that doesn't generate duplicate code in the blockchain. Both of these are improvements over the current design of Ethereum, where EVM supports dynamic code generation.

Contract Initialization A special private function called `@init` can be defined and will be executed at contract creation time. This function typically contains initialization code, e.g. initialization of the account storage, etc. This function is not callable and it can only be invoked at contract creation time. This way, it is guaranteed that the state of an already deployed contract cannot be reset maliciously by invoking initialization code after contract creation. The above is another weakness of the current Ethereum design that has been exploited successfully in the past [Breidenbach et al. 2017].

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 an 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.

```
define public @factorial(%n) {
  // ensure that %n is larger than or equal to 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()
}
```

Listing 3. IELE function for factorial

Control Flow The function body contains code organized in labeled blocks. The execution falls through from the last instruction of a block to the first of the next one, or jumps to the start of a specific block when encountering a branch instruction. 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 dynamic control flow of EVM, where jumps amount to pushing a possibly computed number on the stack and then jumping to it regarded as an address. The IELE design ensures a statically known control flow graph and thus makes automated formal verification easier.

Function Calls A public function can be invoked either manually by an Ethereum end user posting an appropriate transaction, or dynamically by another executing contract using IELE's account call instruction, `call .. at`. The address of the callee account 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 account to the callee account. IELE defines a simple call/return convention as part of the language: called functions expect a specific number of arguments (the number of formal arguments in the function definition) and return a specific number of return values (the number of return values at a `ret` instruction plus an exit status value). If an ABI error occurs (e.g. incorrect number of arguments passed, function not found or not public, etc), a corresponding erroneous exit status is returned. For reference, EVM does not have a well-defined call/return convention: The caller simply sends an arbitrary bytestream containing the call argument values. There is an externally defined ABI that most EVM compilers follow when compiling high-level functions to EVM, but it is not enforced by the language definition. Moreover EVM does not have a notion of callable function, instead the code execution always starts at the contract's first instruction when an account call is received. The higher abstraction level of IELE's calling convention and its more structured design makes IELE contracts more readable and less tedious to formally verify.

The EVM language 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. On the other hand, 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 [Steiner 2017], in IELE we decided to completely drop the `delegatecall` functionality.

Deposit Handler A special public function called `@deposit` can be defined for every IELE contract and it is invoked whenever the account associated with the contract receives a payment in Ether that is not accompanied with a specific function call. Note that a contract can forbid such payments by refusing to define a `@deposit` function.

Local Memory An additional form of storage, called local memory, is available during execution of a function's body. This is a 2^{256} sparse map of arbitrary-sized byte buffers. Unlike the account storage that persists throughout a contract's lifetime, the local memory contents have a lifetime equal to a single account call and return, and are reset to zero whenever a public function is invoked from outside the contract. The contents of the local memory can be accessed through the dedicated `load` and `store` instructions.

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 and/or network state, cryptographic functions, etc.

IELE is a register-based language: Instructions operate on and store their output in virtual registers. An infinite amount of virtual registers is available to a IELE function, but the actual (finite) number of virtual registers used by the function can be determined statically by counting the different register names used in the function's 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, formal verification tools for IELE will not need to reason about the size of the operation stack, another tedious requirement for verifying EVM programs.

3.4 IELE Datatypes

We use arbitrary-precision signed integer as the main datatype in IELE: All virtual registers and account storage locations hold values of this type. They can also be stored in and loaded from the local memory with dedicated instructions. Arbitrary-precision arithmetic removes arithmetic overflows thus making specification and formal verification less tedious, as well as making attacks that exploit arithmetic overflow bugs not viable. At least one such attack has been reported against an EVM contract [PeckShield 2018].

3.5 Formal Semantics as Implementation

The IELE language has its reference implementation automatically derived from its formal specification. This approach is enabled by the K framework [Rosu and Serbanuta 2010]: we provide the IELE formal specification as a K semantic definition and we get auto-generated and correct by construction tools such as parser, interpreter, deductive verifier, and more. So far, K has been used successfully to formalize existing languages [Bogdanas and Rosu 2015; Ellison and Rosu 2012; Filaretti and Maffeis 2014; Park et al. 2015]. In designing IELE, we followed the reverse path: Instead of implementing the language based on a loose specification first, and then attempt to capture its specification in a formal way, *we designed the language by directly giving its formal specification and we used K to derive its implementation.* This approach allows us to:

- have a strict and formal specification of the language,
- easily prototype different specification choices during design,
- and get an automatically derived and correct by construction reference implementation for the language.

In designing IELE there was no trade-off between strictness of specification and ease of prototyping: any modification in the specification is automatically reflected in the reference implementation and can be tested at the spot. Moreover, testing the implementation increases confidence about the formal specification at the same time: there are no bugs introduced by the implementation, and any bugs discovered during testing can be traced back to the formal specification.

IELE is the first real-world language to our knowledge, that is designed and implemented using formal semantics, with a zero gap between the formal specification and the implementation. Until now, real-world languages were either given semantics post-mortem, or had only been given an untested semantics apriori followed by an adhoc implementation; not surprisingly, errors were found in the language specification after the language was released (see for example the formal definition of SML [Milner et al. 1997]). With IELE, we are demonstrating that the PL semantics technology has reached the point where languages can be designed and implemented by simply giving them a formal semantics.

3.6 IELE Formal Semantics Overview

The formal semantics of IELE [Anonymous 2018a] is spread among several files: `iele-syntax.md` contains a quick introduction to the various features of the language along with their syntactical definitions. `iele.md` contains the semantics of the various language features as well as 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 that is also 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 various 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 an IELE contract and its contents.

```
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, ","}
```

Listing 4. K syntax productions for IELE contracts

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).

```
<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>    // The name of the function
      <nparams>   0       </nparams>   // The number of parameters of the function
      <jumpTable> .Map    </jumpTable> // Map from jump labels to blocks
      <nregs>     0       </nregs>     // Number of registers used by this function
      <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
```

Listing 5. Configuration for local state

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 smart contract that is associated with the active account. 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 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 that are alive in the current call stack.

2) The network state contains information about the whole 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 Ethereum accounts on the network.

```
<network>
  <accounts>
    <account multiplicity="*" type="Map">
      <acctID>   0          </acctID>     // ID of account
      <balance>  0          </balance>    // Balance of funds in account
      <code>     #emptyCode </code>       // Disassembled contract of account
      <storage>  .Map       </storage>    // Permanent storage of account
      <nonce>    0          </nonce>      // Nonce of account
    </account>
  </accounts>

  // ... more cells ...

</network>
```

Listing 6. Configuration for network state

The accounts contains one account for each active 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 smart contract code associated with the account. The storage cell maps storage addresses to their current contents for the account's storage. The nonce cell contains a monotonically increasing integer value that counts the number of transactions performed by this account.

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 generate 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 uses these rules to match patterns found in the left-hand side of rules with the contents of the current configuration, and rewrite it according to the right-had side.

Traditionally, the k cell of the configuration drives the execution by containing at any time a list of execution steps to be matched and rewritten. In the case of the IELE semantics, we define a set of internal operators that represent different execution steps and maintain a list of such oprators inside the k cell during execution.

As an example, Listing 7 shows the rules that specify the behavior of 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>
```

Listing 7. Rules for the div instruction

The syntax production defines the internal operator #exec, which represents the execution of a single IELE instruction. Both the 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 that zero, as specified in the requires clause of the rule. In this case the top of the k cell

is rewritten to another internal operator, `#load`, that loads the result of the division to the virtual register in the left-hand side of the division instruction. The `/Int` and `=/=Int` operators are K built-in operators for arbitrary-precision signed integers.

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 account that initiated the transaction.

In both the above rules, the ellipses (`...`) is a K syntax for patterns that matches the rest of the `k` cell, which is a list of internal operators.

The rules shown in Listing 8 specify the behavior of the `#load` internal operator that is used to store values into 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.

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

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

The first rule accesses the current register file found 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.

Note that these rules contain patterns over several configuration cells, but we only need to include in the rule the cells that we want to match. K can automatically infer and fill in the rest of the configuration to generate a full pattern. Also note that a rule can contain multiple rewrites that all resolve simultaneously.

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

```
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
```

Listing 9. Rules for the `intSize` operator

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.

3.7 IELE Gas Model

The 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, in the cryptocurrency of the blockchain. When the transaction is incorporated into the blockchain the sender pays the miner for the gas actually used, according to the price. If the maximum gas is exceeded then the sender pays for all the gas but the effects of executing the code are rolled back. Separating gas costs and the price of gas allows adjustment for varying value of the cryptocurrency and of computation, or for the willingness of miners to run longer computations.

The gas costs are generally meant to be proportional to computation time, with additional charges for the amount of memory used during contract execution, and larger costs for any operations changing persistent state that needs to be stored by every full participant in the blockchain. Computation costs must never be asymptotically higher than gas cost, otherwise the difference allows denial of service attacks. Computation costs should also be chosen so that costs are rarely an excessive overestimate, so that miners can decide whether a transaction is worth running from the declared gas price rather than also needing a detailed prediction of execution time vs. gas cost. Several changes to the EVM gas model have been made when operations that had been assumed linear were discovered to have actually quadratic execution time, or instructions assumed constant-time were discovered to grow with program size. In IELE we should also 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 (a IELE implementation will likely execute using efficient native-code implementations of good algorithms).

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 [Free Software Foundation 2018] 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 10.

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

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

Listing 10. Gas formulas for bitwise and and `iszero`

More complicated formulas were needed for multiplication and related operations such as 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 [Karatsuba and Ofman 1962] 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 following definition 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.

```
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
```

Listing 11. Approximating $n^{log_3 2}$

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):

```
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
```

Listing 12. Approximating gas costs for exponentiation

The gas cost for memory is based on peak memory consumption. Local memory is used by values in registers and in memory, and also in stack frames for contract-local function calls. RAM consumption is not commensurate with CPU time, so 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 memory-based 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. In this case the peak memory is increased if needed to make room for the largest result size, the gas charged for an increased peak, and then count of current memory adjusted downward if the actual result is smaller.

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:

```
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 ]}:>Int))
     </peakMemory>
```

Listing 13. Updating peak memory and assesing gas costs

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 14.

```
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)
```

Listing 14. Gas formulas for `or`, `add`, and `exp`

For storage based costs, operations which transfer cryptocurrency, create new accounts, or modify persistent storage have high costs, reflecting that these permanently increase the size of blockchain history. Operations which query persistent state also have elevated costs, since they might require the computer executing the contract to retrieve the 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. We believe this can replace most cases where EVM programs would use the dangerous `delegatecall` instruction purely to reduce the cost of creating many smart contracts sharing behavior.

For example, the cost `Gbalance` of looking up the balance of an account is over 100 times the base 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.

4 ISOLC: THE SOLIDITY TO IELE COMPILER

Solidity [Ethereum 2018b] is a popular high-level language for smart contracts. It is a statically-typed language with object oriented features, such as inheritance, and user-defined structure types. A compiler from Solidity to EVM, *solc*, is commonly used by smart contract developers to generate EVM code that can be uploaded in an Ethereum blockchain. Although the Solidity language, and its EVM compiler for that matter, are in early development, they are widely used: As of July 2018, more than 30000 accounts of the Ethereum network are associated with code compiled down from Solidity [Etherscan 2018b], including accounts that handle as much as 400M USD worth of Ether [Multisig 2018].

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* that can be found here [Anonymous

2018b]. Our compiler reuses much of the front-end of *solc* and includes a new backend to generate IELE code. We used *isolc* to compile the more than 900 Solidity smart contracts of the *solc* test suite to IELE, including real-world contracts, such as a multi-signature wallet and an ERC20 compliant token.

At the time of writing, the compiler supports most of the lastest 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 are not expressible in IELE. The additional features are taking advantage of some new IELE features and/or serve as replacements for the unsupported features. In the following we discuss representative examples of the above categories.

Low-level calls Solidity supports a low-level account call built-in that has similar semantics with an EVM call: The caller can use the built-in to send a arbitrary bytestream of data to the callee account. The call/return ABI of IELE is not compatible with this low-level operation, hence for Solidity code to be able to compile to IELE, such calls should be eliminated and replaced by normal Solidity account calls to functions that accept an argument of type `bytes`, which is the Solidity type for bytestreams.

delegatecall is another low-level Solidity built-in for account calls that exposes the corresponding EVM functionality. As discussed in section 3.2, IELE dropped this functionality for security reasons (see section 6.4), and in turn our Solidity compiler does not support the built-in.

Inline EVM assembly Solidity supports inline snippets of EVM assembly, which is naturally non-expressible in IELE. To reduce the necessity for inline EVM assembly in Solidity programs, we added various built-ins to the global namespace of Solidity that give access to functionality that was previously only available through inline EVM code. For example we add Solidity built-ins for the zero-knowledge proof operations `ecadd`, `ecmull`, and `ecpairing`.

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. We adjust the type casting and type inference semantics of Solidity appropriately: The default inferred type for some value is the arbitrary-precision type only if inferring any fixed-width integer would lose information. Also any casting that would lose information is not implicitly allowed.

The full list of feature differences can be found in the `READ-ME-IELE-SUPPORT.md` file of the *isolc* [Anonymous 2018b] repository. We also provide examples of how to avoid/replace these features in exisiting Solidity smart contracts, so that they can be compiled to IELE.

5 IELE VM AND INTEGRATION WITH MANTIS

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 an IELE-based network, when IELE code is invoked and/or deployed as part of a transaction, an IELE VM is needed to execute the code and update 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 checker. We have also integrated our IELE VM with the Mantis Ethereum client [IOHK 2018b]. Using Mantis we have successfully deployed and executed IELE smart contracts on a Ethereum-based blockchain.

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 within the IELE VM, 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.

6 EVALUATION

In this section we evaluate the following claims:

- The formal specification of the IELE language is complete and correct. As an extension, the IELE VM that is derived from the specification is also complete and correct.
- The Solidity to IELE compiler is complete and correct.
- The design approach of specification as the implementation is a realistic one in terms of effort required.
- The IELE VM has acceptable performance.
- The design of the IELE language prevents various exploits found in EVM contracts that have cost millions of USD in the past.

6.1 Completeness and Correctness

To demonstrate that the IELE language specification, the derived IELE VM, and the Solidity to IELE compiler are complete and correct, 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 [Ethereum 2018c] as our supply of Solidity contracts. This test suite contains a total of 1971 test cases. The same test suite is used to test the production version of the Solidity to EVM compiler. For example, one of the real-world contracts contained in the test suite, a multisig wallet, is currently associated (in its compiled EVM version) with the 5th largest account in the Ethereum network, managing at the time of writing more than 400M USD [Multisig 2018]. We chose the Solidity to EVM compiler's test suite for the additional advantage that it contains many unit and regression tests that exercise corner cases in the compiler and the specification. We modify any tests that use Solidity features not supported when compiling to IELE (see section 4).

Our pipeline succeeds in 100% of the tests, meaning that in all cases the Solidity to IELE compiler can generate a IELE contract that can in turn be successfully deployed and executed using the IELE VM. This fact provides high confidence on the correctness and completeness of both the Solidity to IELE compiler and the IELE specification.

6.2 Design and Implementation Effort

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 [Hildenbrandt et al. 2018] 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.

6.3 Performance evaluation

We evaluate two aspects of the IELE implementation: the binary size of IELE programs 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 respectively.

Binary Size We compiled 963 Solidity unit tests and 5 large contracts to both EVM and IELE and measured the size in bytes of the resulting binary IELE and EVM programs. These are all the tests in the Solidity test suite that did not require any modification to compile to EVM and IELE. 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 same 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 a lot of stack manipulation opcodes like PUSH found in EVM programs. However our compiler at this time uses a very naive code generation algorithm and does not perform any optimizations and as a result generates more code for larger input programs. We believe that the IELE the binary size can 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 64MB 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 when serving the same transaction. For the large contracts, the same ratio is 1.18, meaning that the IELE VM is on average 1.18 times faster than KEVM when serving the same transaction. KEVM has been shown to be one order of magnitude slower than the C++ reference implementation of the EVM, cpp-ethereum [Ethereum 2018a].

This speedup over KEVM is due to our optimization of the IELE specification for execution, using a number of standard practices. We did this by profiling the semantics to find hotspots and improving their performance in different ways specific to each case. In some cases, a new data structure was introduced which had better asymptomatic complexity. In others, the algorithm was rewritten in order to minimize use of costly operations or to fix bugs which increased the asymptomatic performance. Where excessive stack usage was discovered, functions were rewritten to become tail recursive. It is also important to note that the IELE semantics has reused parts of the EVM K specification, especially when it comes to low-level data structures and functionality. Hence, many of the above optimizations are on top of the inherited EVM functionality.

These results show that the IELE compiler and the IELE VM have competitive performance with the current state of the art. These improvements in performance should be regarded as a bonus for a cleaner design and the choice of a register-based rather than stack-based architecture, because as explained in section 3, the main forces that drove the design of IELE were not related to performance. Nevertheless, it is nice to know that performance is not only not hurt, but improved.

6.4 Security

In the following, we review three famous and costly recent exploits of contracts deployed in the Ethereum blockchain. We discuss the exploit and how it could be avoided in an IELE-based

blockchain.

Parity Wallet Freeze (Nov 2017) [Steiner 2017] 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, all 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. It is important to note that by nature of the blockchain, accepted transactions are irreversible, meaning that it is not possible for Parity to roll back the deletion transaction or for users to roll back the wallet code deployment transactions.

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 of the IELE language, hence making an attack such as this not possible.

Parity Wallet Breach (Jul 2017) [Breidenbach et al. 2017] 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. So, the attackers reinitialized ownership of various wallet accounts to themselves which in turn gave them the ability to withdraw those accounts' balances. It is estimated that 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 an EVM-based blockchain. Contract deployment and initialization in IELE can be done in a single step: the `@init` function is called at contract creation time and can contain storage initialization code. Moreover the function is private, meaning it cannot be invoked by other accounts through an account call.

The DAO attack (Jun 2016) [Daian 2016] A simplified version of the vulnerable contract in the DAO attack is shown in Listing 15. 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 an account call back to the caller account, `withdrawBalance` uses the low-level `call` of Solidity (see section 4). 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 mapping.

```
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;
   }
}
```

Listing 15. Solidity contract that showcases the DAO exploit

The attacker in the DAO attack exploited a similar vulnerability and stole more than 150M USD from the contract. This attack prompted for the first time a hard fork in the Ethereum blockchain [del Castillo 2016].

Of course, the exploit above is eliminated by reordering the sending of Ether and the mapping update. However, there is another way to eliminate the exploit: 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 4, the Solidity to IELE compiler does not support the low-level `call`, hence making this particular case of re-entrancy inexpressible.

7 CONCLUSION

In this paper, we presented IELE, a new intermediate-level language for smart contracts. The full formal specification of IELE is available and is the main artifact of our design. We used the K framework to automatically generate a faithful implementation of the language that we evaluated to be in par with the state of the art in the world of smart contract virtual machines.

IELE is designed as an improvement over the current virtual machine used to execute smart contracts, EVM. As adopters of KEVM, the executable formalization of EVM, we sought to design a language that avoids the drawbacks of EVM that KEVM made apparent: insecure low-level features (such as `delegatecall`, overflows, undefined call/return ABI) and tedious formal verification.

IELE is a secure, well-defined, human-readable, language appropriate as the compilation target for high-level languages.

By formally specifying the complete IELE language and generating its virtual machine directly from the speacification, we believe we have significantly raised 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.

We also presented a compiler from the popular Solidity smart contract language to IELE, as well as a fully functional client for the Ethereum blockchain that is based on Mantis and powered by the IELE virtual machine. These tools allowed us to deploy and execute IELE contracts, compiled from real-world contracts, in an Ethereum blockchain. We look forward to using IELE in a real-world blockchain in the near future, making IELE the first practical language to be developed directly as a formal specification.

REFERENCES

Anonymous. 2018a. The formal semantics for IELE — source code. (2018). Supplemental material.

Anonymous. 2018b. The Solidity to IELE compiler — source code. (2018). Supplemental material.

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

Denis Bogdanas and Grigore Rosu. 2015. K-Java: A Complete Semantics of Java. In *Proceedings of the 42nd Symposium on Principles of Programming Languages (POPL'15)*. ACM, 445–456. https://doi.org/10.1145/2676726.2676982

Lorenz Breidenbach, Phil Daian, Ari Juels, and Emin Gün Sirer. 2017. An In-Depth Look at the Parity Multisig Bug. (2017). http://hackingdistributed.com/2017/07/22/deep-dive-parity-bug/

V. Buterin and Ethereum Foundation. 2013. Ethereum White Paper. https://github.com/ethereum/wiki/wiki/White-Paper. (2013).

CoinMarketCap. 2018. Ethereum Market Capacity. https://coinmarketcap.com/currencies/ethereum. (2018).

Phil Daian. 2016. DAO Attack. (2016). http://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/

Michael del Castillo. 2016. Ethereum Executes Blockchain Hard Fork to Return DAO Funds. (2016). http://www.coindesk.com/ethereum-executes-blockchain-hard-fork-return-dao-investor-funds/

Chucky Ellison and Grigore Rosu. 2012. An Executable Formal Semantics of C with Applications. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'12)*. ACM, 533–544. https://doi.org/10.1145/2103656.2103719

Ethereum. 2018a. Ethereum C++ Client. (2018). https://github.com/ethereum/cpp-ethereum

Ethereum. 2018b. Solidity Documentation. (2018). http://solidity.readthedocs.io

Ethereum. 2018c. Solidity Github Project. (2018). https://github.com/ethereum/solidity

Etherscan. 2018a. Ethereum Transaction Growth. https://etherscan.io/chart/tx. (2018).

Etherscan. 2018b. Ethereum Verified Contract Sources. https://etherscan.io/contractsVerified. (2018).

Daniele Filaretti and Sergio Maffeis. 2014. An Executable Formal Semantics of PHP. In *ECOOP 2014 – Object-Oriented Programming*, Richard Jones (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 567–592.

Inc. Free Software Foundation. 2018. GNU MP Documentation. (2018). https://gmplib.org/manual

Chris Hathhorn, Chucky Ellison, and Grigore Rosu. 2015. Defining the Undefinedness of C. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)*. ACM, 336–345. https://doi.org/10.1145/2813885.2737979

Everett Hildenbrandt, Manasvi Saxena, Xiaoran Zhu, Nishant Rodrigues, Philip Daian, Dwight Guth, Brandon Moore, Yi Zhang, Daejun Park, and Grigore Roşu. 2018. KEVM: A Complete Semantics of the Ethereum Virtual Machine. In *Computer Security Foundations Symposium*.

IOHK. 2018a. Input Output HK. (2018). https://iohk.io

IOHK. 2018b. Mantis Ethereum Classic Client. (2018). https://iohk.io/blog/mantis-ethereum-classic-beta-release

A. Karatsuba and Yu. Ofman. 1962. Multiplication of many-digital numbers by automatic computer. In *Dokl. Akad. Nauk SSSR*, Vol. 145. 293–294.

KEVM. 2018. Jello Paper. (2018). https://jellopaper.org/

Eleftherios Kokoris Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. 2016. Enhancing Bitcoin Security and Performance with Strong Consistency via Collective Signing. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, Austin, TX, 279–296. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/kogias

Chris Lattner and Vikram Adve. 2004. 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 (CGO '04)*. IEEE Computer Society, Washington, DC, USA, 75–. http://llvm.org

Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. 1997. *The definition of standard ML : revised.* MIT Press, Cambridge, Mass.

Multisig. 2018. Multi-signature Wallet Ethereum Account. https://etherscan.io/address/0xab7c74abc0c4d48d1bdad5dcb26153fc8780f83e#code. (2018).

Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. (2008). https://bitcoin.org/bitcoin.pdf

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

Bobby Ong, Teik Ming Lee, Guo Li, and David LEE Kuo Chuen. 2015. Evaluating the Potential of Alternative Cryptocurrencies. In *Handbook of Digital Currency*, David Lee Kuo Chuen (Ed.). Academic Press, San Diego, 81 – 135. https://doi.org/10.1016/B978-0-12-802117-0.00005-9

Daejun Park, Andrei Stefanescu, and Grigore Rosu. 2015. KJS: A Complete Formal Semantics of JavaScript. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)*. ACM, 346–356. https://doi.org/10.1145/2737924.2737991

PeckShield. 2018. 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.

Grigore Rosu and Traian Florin Serbanuta. 2010. An Overview of the K Semantic Framework. *Journal of Logic and Algebraic Programming* 79, 6 (2010), 397–434. http://kframework.org

Grigore Rosu, Andrei Stefanescu, Stefan Ciobaca, and Brandon M. Moore. 2013. One-Path Reachability Logic. In *Proceedings of the 28th Symposium on Logic in Computer Science (LICS'13)*. IEEE, 358–367.

Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. 2014. Zerocash: Decentralized Anonymous Payments from Bitcoin. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy (SP '14)*. IEEE Computer Society, Washington, DC, USA, 459–474. https://doi.org/10.1109/SP.2014.36

Jasmine Solana. 2017. $500K hack challenge backfires on blockchain lottery SmartBillions. (2017). https://calvinayre.com/2017/10/13/bitcoin/500k-hack-challenge-backfires-blockchain-lottery-smartbillions/

Jutta Steiner. 2017. Security is a process: A Postmortem on the Parity Multi-Sig Library Self-Destruct. (2017). http://goo.gl/LBh1vR.

Nick Szabo. 1994. 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

The Ethereum Foundation. 2018a. ERC20 Token Standard. (2018). https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20-token-standard.md

The Ethereum Foundation. 2018b. Multi-Signature Wallet. (2018). https://github.com/ConsenSys/MultiSigWallet

Shawn Wilkinson, Tome Boshevski, Josh Brandoff, and Vitalik Buterin. 2014. Storj A Peer-to-Peer Cloud Storage Network. (2014).

Gavin Wood. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151 (2014), 1–32.