Foundations and Tools for the Static Analysis of Ethereum smart contracts

Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind

TU Wien
{ilya.grishchenko,matteo.maffei,clara.schneidewind}@tuwien.ac.at

Abstract. The recent growth of the blockchain technology market puts its main cryptocurrencies in the spotlight. Among them, Ethereum stands out due to its virtual machine (EVM) supporting smart contracts, i.e., distributed programs that control the flow of the digital currency Ether. Being written in a Turing complete language, Ethereum smart contracts allow for expressing a broad spectrum of financial applications. The price for this expressiveness, however, is a significant semantic complexity, which increases the risk of programming errors. Recent attacks exploiting bugs in smart contract implementations call for the design of formal verification techniques for smart contracts. This, however, requires rigorous semantic foundations, a formal characterization of the expected security properties, and dedicated abstraction techniques tailored to the specific EVM semantics. This work will overview the state-of-the-art in smart contract verification, covering formal semantics, security definitions, and verification tools. We will then focus on EtherTrust [1], a framework for the static analysis of Ethereum smart contracts which includes the first complete small-step semantics of EVM bytecode, the first formal characterization of a large class of security properties for smart contracts, and the first static analysis for EVM bytecode that comes with a proof of soundness.

1 Introduction

Blockchain technologies promise secure distributed computations even in absence of trusted third parties. The core of this technology is a distributed ledger that keeps track of previous transactions and the state of each account, and whose functionality and security is ensured by a careful combination of incentives and cryptography. Within this framework, software developers can implement sophisticated distributed, transaction-based computations by leveraging the scripting language offered by the underlying cryptocurrency. While many of these cryptocurrencies have an intentionally limited scripting language (e.g., Bitcoin [2]), Ethereum was designed from the ground up with a quasi Turing-complete language[1]. Ethereum programs, called *smart contracts*, have thus found a variety of appealing use cases, such as auctions [3], data management systems [4], financial contracts [5], elections [6], trading platforms [7,8], permission management [9] and verifiable cloud computing [10], just to mention a few. Given

[1] While the language itself is Turing complete, computations are associated with a bounded computational budget (called gas), which gets consumed by each instruction thereby enforcing termination.

their financial nature, bugs and vulnerabilities in smart contracts may lead to catastrophic consequences. For instance, the infamous DAO vulnerability [11] recently led to a 60M$ financial loss and similar vulnerabilities occur on a regular basis [12,13]. Furthermore, many smart contracts in the wild are intentionally fraudulent, as highlighted in a recent survey [14].

A rigorous security analysis of smart contracts is thus crucial for the trust of the society in blockchain technologies and their widespread deployment. Unfortunately, this task is quite challenging for various reasons. First, Ethereum smart contracts are developed in an ad-hoc language, called Solidity, which resembles JavaScript but features specific transaction-oriented mechanisms and a number of non-standard semantic behaviours, as further described in this paper. Second, smart contracts are uploaded on the blockchain in the form of Ethereum Virtual Machine (EVM) bytecode, a stack-based low-level code featuring dynamic code creation and invocation and, in general, very little static information, which makes it extremely difficult to analyze.

Our Contributions This work overviews the existing approaches taken towards formal verification of Ethereum smart contracts and discusses EtherTrust, the first sound static analysis tool for EVM bytecode. Specifically, our contributions are

- A survey on recent theories and tools for formal verification of Ethereum smart contracts including a systematization of existing work with an overview of the open problems and future challenges in the smart contract realm.
- An illustrative presentation of the small-step semantics presented by [15] with special focus on the semantics of the bytecode instructions that allow for the initiation of internal transactions. The subtleties in the semantics of these transactions have shown to form an integral part of the attack surface in the context of Ethereum smart contracts.
- A review of an abstraction based on Horn clauses for soundly over-approximating the small-step executions of Ethereum bytecode [1].
- A demonstration of how relevant security properties can be over-approximated and automatically verified using the static analyzer EtherTrust [1] by the example of the single-entrancy property defined in [15].

Outline The remainder of this paper is organized as follows. § 2 briefly overviews the Ethereum architecture, § 3 reviews the state of the art in formal verification of Ethereum smart contracts, § 4 revisits the Ethereum small-step semantics introduced by [15], § 5 presents the single-entrancy property for smart contracts as defined by [15], § 6 discusses the key ideas of the first sound static analysis for Ethereum bytecode as implemented in EtherTrust [1], § 7 shows how reachability properties can automatically be checked using EtherTrust, and § 8 concludes summarizing the key points of the paper.

2 Background on Ethereum

In the following we will shortly overview the mechanics of the cryptocurrency Ethereum and its built-in scripting language EVM bytecode.

2.1 Ethereum

Ethereum is a cryptographic currency system built on top of a blockchain. Similar to Bitcoin, network participants publish transactions to the network that are then grouped into blocks by distinct nodes (the so called *miners*) and appended to the blockchain using a proof of work (PoW) consensus mechanism. The state of the system – that we will also refer to as *global state* – consists of the state of the different accounts populating it. An account can either be an external account (belonging to a user of the system) that carries information on its current balance or it can be a contract account that additionally obtains persistent storage and the contract's code. The account's balances are given in the subunit *wei* of the virtual currency *Ether*.[2]

Transactions can alter the state of the system by either creating new contract accounts or by calling an existing account. Calls to external accounts can only transfer Ether to this account, but calls to contract accounts additionally execute the code associated to the contract. The contract execution might alter the storage of the account or might again perform transactions – in this case we talk about *internal transactions*.

The execution model underlying the execution of contract code is described by a virtual state machine, the *Ethereum Virtual Machine* (EVM). This is *quasi Turing complete* as the otherwise Turing complete execution is restricted by the upfront defined resource *gas* that effectively limits the number of execution steps. The originator of the transaction can specify the maximal gas that should be spent for the contract execution and also determines the gas price (the amount of wei to pay for a unit of gas). Upfront, the originator pays for the gas limit according to the gas price and in case of successful contract execution that did not spend the whole amount of gas dedicated to it, the originator gets reimbursed with gas that is left. The remaining wei paid for the used gas are given as a fee to a beneficiary address specified by the miner.

2.2 EVM bytecode

Contracts are delivered and executed in *EVM bytecode* format – an Assembler like bytecode language. As the core of the EVM is a stack-based machine, the set of instructions in EVM bytecode consists mainly of standard instructions for stack operations, arithmetics, jumps and local memory access. The classical set of instructions is enriched with an opcode for the SHA3 hash and several opcodes for accessing the environment that the contract was called in. In addition, there are opcodes for accessing and modifying the storage of the account currently running the code and distinct opcodes for performing internal call and create transactions. Another instruction particular to the blockchain setting is the SELFDESTRUCT code that deletes the currently executed contract - but only after the successful execution of the external transaction.

The execution of each instruction consumes a positive amount of *gas*. The sender of the transaction specifies a gas limit and exceeding it results in an exception that reverts the effects of the current transaction on the global state. In the case of nested transactions, the occurrence of an exception only reverts its own effects, but not those of the calling transaction. Instead, the failure of an internal transaction is only indicated by writing zero to the caller's stack.

[2] One Ether is equivalent to 10^{18} wei.

3 Overview on formal verification approaches

In the following we give an overview on the approaches taken so far in the direction of securing (Ethereum) smart contracts. We distinguish between verification approaches and design approaches. According to our terminology, the goal of verification approaches is to check smart contracts written in existing languages (such as Solidity) for their compliance with a security policy or specification. In contrast, design approaches aim at facilitating the creation of secure smart contracts by providing frameworks for their development: These approaches encompass new languages which are more amenable to verification, provide a clear and simple semantics that is understandable by smart contract developers or allow for a direct encoding of desired security policies. In addition, we count works that aim at providing design patterns for secure smart contracts to this category.

3.1 Verification

In the field of smart contract verification we categorize the existing approaches along the following dimensions: target language (bytecode vs high level language), point of verification (static vs. dynamic analysis methods), provided guarantees (bug-finding vs. formal soundness guarantees), checked properties (generic contract properties vs. contract specific properties), degree of automation (automated verification vs. assisted analysis vs. manual inspection). From the current spectrum of analysis tools, we can find solutions in the following clusters:

Static analysis tools for automated bug-finding. Oyente [16] is a state-of-the-art static analysis tool for EVM bytecode that relies on symbolic execution. Oyente supports a variety of pre-defined security properties, such as transaction order dependency, timestamp dependency, and reentrancy that can be checked automatically. However, Oyente is not striving for soundness nor completeness. This is on the one hand due to the simplified semantics that serves as foundation of the analysis [15]. On the other hand, the security properties are rather syntactic or pattern based and are lacking a semantic characterization. Recently, Zhou et al. proposed the static analysis tool SASC [17] that extends Oyente by additional patterns and provides a visualization of detected risks in the topology diagram of the original Solidity code.

Majan [18] extends the approach taken in Oyente to trace properties that consider multiple invocations of one smart contract. As Oyente, it relies on symbolic execution that follows a simplified version of the semantics used in Oyente and uses a pattern-based approach for defining the concrete properties to be checked. The tool covers safety properties (such as prodigality and suicidality) and liveness properties (greediness). As for Oyente, the authors do not make any security claims, but consider their tool a 'bug catching approach'.

Static analysis tools for automated verification of generic properties. In contrast to the aforementioned class of tools, this line of research aims at providing formal guarantees for the analysis results.

A recently published work is the static analysis tool ZEUS [19] that analyzes smart contracts written in Solidity using symbolic model checking. The analysis proceeds by translating Solidity code to an abstract intermediate language that again is translated to LLVM bitcode. Finally, existing symbolic model checking tools for LLVM bitcode are leveraged for checking generic security properties. ZEUS consequently only allows for analyzing contracts whose Solidity source code is made available. In addition, the semantics of the intermediate language cannot easily be reconciled with the actual Solidity semantics that is determined by its translation to EVM bytecode. This is as the semantics of the intermediate language by design does not allow for the revocation of the global system state in the case of a failed call – which however is fundamental feature of Ethereum smart contract execution.

Other tools proposed in the realm of automated static analysis for generic properties are Securify [20], Mythril [21] and Manticore [22] (for analysing bytecode) and SmartCheck [23] and Solgraph [24] (for analyzing Solidity code). These tools however are not accompanied by any academic paper so that the concrete analysis goals stay unspecified.

Frameworks for semi-automated proofs for contract specific properties. Hirai [25] formalizes the EVM semantics in the proof assistant Isabelle/HOL and uses it for manually proving safety properties for concrete contracts. This semantics, however, constitutes a sound over-approximation of the original semantics [26]. Building on top of this work, Amani et al. propose a sound program logic for EVM bytecode based on separation logics [27]. This logic allows for semi-automatically reasoning about correctness properties of EVM bytecode using the proof assistant Isabelle/HOL.

Hildebrandt et al. [28] define the EVM semantics in the $\mathbb{K}$ framework [29] – a language independent verification framework based on reachability logics. The authors leverage the power of the $\mathbb{K}$ framework in order to automatically derive analysis tools for the specified semantics, presenting as an example a gas analysis tool, a semantic debugger, and a program verifier based on reachability logics. The derived program verifier still requires the user to manually specify loop invariants on the bytecode level.

Bhargavan et al. [30] introduce a framework to analyze Ethereum contracts by translation into F*, a functional programming language aimed at program verification and equipped with an interactive proof assistant. The translation supports only a fragment of the EVM bytecode and does not come with a justifying semantic argument.

Dynamic monitoring for predefined security properties. Grossman et al. [31] propose the notion of effectively callback free executions and identify the absence of this property in smart contract executions as the source of common bugs such as reentrancy. They propose an efficient online algorithm for discovering executions violating effectively callback freeness. Implementing a corresponding monitor in the EVM would guarantee the absence of the potentially dangerous smart contract executions, but is not compatible with the current Ethereum version and would require a hard fork.

A dynamic monitoring solution compatible with Ethereum is offered by the tool DappGuard [32]. The tool actively monitors the incoming transactions to a smart contract and leverages the tool Oyente [16], an own analysis engine and a simulation of the transaction on the testnet for judging whether the incoming transaction might cause a (generic) security violation (such as transaction order dependency). If a transaction is considered harmful, a counter transaction (killing the contract or performing some other fixes) is made. The authors claim that this transaction will be mined with high probability before the problematic one. Due to this uncertainty and the bug-finding tools used for evaluation of incoming transactions, this approach does not provide any guarantees.

3.2 Design

The current research on secure smart contract design focuses on the following four areas: high-level programming languages, intermediate languages (for verification), security patterns for existing languages and visual tools for designing smart contracts.

High-level languages. One line of research on high-level smart contract languages concentrates on the facilitation of secure smart contract design by limiting the language expressiveness and enforcing strong static typing discipline. Simplicity [33] is a typed functional programming language for smart contracts that disallows loops and recursion. It is a general purpose language for smart contracts and not tailored to the Ethereum setting. Simplicity comes with a denotional semantics specified in Coq that allows for reasoning formally about Simplicity contracts. As there is no (verified) compiler to EVM bytecode so far, such results don't carry over to Etherum smart contracts. In the same realm, Pettersson and Edström [34], propose a library for the programming language Idris that allows for the development of secure smart contracts using dependent and polymorphic types. They extend the existing Idris compiler with a generator for Serpent code (a Python-like high-level language for Ethereum smart contracts). This compiler is a proof of concept and fails in compiling more advanced contracts (as it cannot handle recursion). In a preliminary work, Coblenz et al. [35] propose Obsidian, an object-oriented programming language that pursues the goal of preventing common bugs in smart contracts such as reentrancy. To this end, Obsidian makes states explicit and uses a linear type system for quantities of money.

Another line of research focuses on designing languages that allow for encoding security policies that are dynamically enforced at runtime. A first step in this direction is sketched in the preliminary work on Flint [36], a type-safe, capabilities-secure, contract-oriented programming language for smart contracts that gets compiled to EVM bytecode. Flint allows for defining caller capabilities restricting the access to security sensitive functions. These capabilities shall be enforced by the EVM bytecode created during compilation. But so far, there is only an extended abstract available.

In addition to these approaches from academia, the Ethereum foundation currently develops the high-level languages Viper [37] and Bamboo [38]. Furthermore, the Solidity compiler used to support a limited export functionality to the intermediate language WhyML [39] allowing for a pre/post condition style reasoning on Solidity code by leveraging the deductive program verification platform Why3 [40].

Intermediate languages. The intermediate language Scilla [41] comes with a semantics formalized in the proof assistant Coq and therefore allows for a mechanized verification of Scilla contracts. In addition, Scilla makes some interesting design choices that might inspire the development of future high level languages for smart contracts: Scilla provides a strict separation not only between computation and communication, but also between pure and effectful computations.

Security patterns. Wöhrer [42] describes programming patterns in Solidity that should be adapted by smart contract programmers for avoiding common bugs. These patterns encompass best coding practices such as performing calls at the end of a function, but also off-the-self solutions for common security bugs such as locking a contract for avoiding reentrancy or the integration of a mechanism that allows the contract owner to disable sensitive functionalities in the case of a bug.

Tools. Mavridou and Laszka [43] introduce a framework for designing smart contracts in terms of finite state machines. They provide a tool with a graphical editor for defining contract specifications as automata and give a translation of the constructed finite state machines to Solidity. In addition, they present some security extensions and patterns that can be used as off-the-shelf solutions for preventing reentrancy and implementing common security challenges such as time constraints and authorization. The approach however is lacking formal foundations as neither the correctness of the translation is proven correct, nor are the security patterns shown to meet the desired security goals.

3.3 Open challenges

Even though the previous section highlights the wide range of steps taken towards the analysis of Ethereum smart contracts, there are still a lot of open challenges left.

Secure compilation of high-level languages. Even though there are several proposals made for new high-level languages that facilitate the design of secure smart contracts and that are more amenable to verification, none of them comes so far with a verified compiler to EVM bytecode. Such a secure compilation however is the requirement for the results shown on high-level language programs to carry over to the actual smart contracts published on the blockchain.

Specification languages for smart contracts. So far, all approaches to verifying contract specific properties focus on either ad-hoc specifications in the used verification framework [25,28,30,27] or the insertion of assertions into existing contract code [39]. For leveraging the power of existing model checking techniques for program verification, the design of a general-purpose contract specification language would be needed.

Study of security policies. There has been no fundamental research made so far on the classes of security policies that might be interesting to enforce in the setting of smart contracts. In particular, it would be compelling to characterize the class of security policies that can be enforced by smart contracts within the existing EVM.

Compositional reasoning about smart contracts. Most research on smart contract verification focuses on reasoning about individual contracts or at most a bunch of contracts whose bytecode is fully available. Even though there has been work observing the similarities between smart contracts and concurrent programs [44], there has been no rigorous study on compositional reasoning for smart contracts so far.

4 Semantics

Recently, Grishchenko et al. [15] introduced the first complete small-step semantics for EVM bytecode. As this semantics serves as a basis for the static analyzer EtherTrust, we will in the following shortly review the general layout and the most important features of the semantics.

4.1 Execution configurations

Before discussing the small-step rules of the semantics, we first introduce the general shape of execution configurations.

Global state. The global state of the Ethereum blockchain is represented as a (partial) mapping from account addresses to accounts. In the case that an account does not exist, we assume it to map to $\bot$. Accounts are composed of a nonce n that is incremented with every other account that the account creates, a balance b, a persistent unbounded storage *stor* and the account's code. External accounts carry an empty code which makes their storage inaccessible and hence irrelevant.

Small-step relation. The semantics is formalized by a small-step relation $\Gamma \vDash S \rightarrow S'$ that specifies how a call stack S representing the state of the execution evolves within one step under the transaction environment Γ. We call the pair (Γ, S) a *configuration*.

Transaction environments. The transaction environment represents the static information of the block that the transaction is executed in and the immutable parameters given to the transaction as the gas prize or the gas limit. These parameters can be accessed by distinct bytecode instructions and consequently influence the transaction execution.

Call stacks. A call stack S is a stack of execution states which represents the state of the overall execution of the initial external transaction. The individual execution states of the stack represent the states of the uncompleted internal transactions performed during the execution. Formally, a call stack is a stack of regular execution states of the form (μ, ι, σ) that can optionally be topped with a halting state $\mathit{HALT}(\sigma, \mathit{gas}, d)$ or an exception state *EXC*. Semantically, halting states indicate regular halting of an internal transaction, exception states indicate exceptional halting, and regular execution states describe the state of internal transactions in progress. Halting and exception states can only occur as top elements of the call stack as they represent terminated internal transactions. Halting states carry the information affecting the callee state such as the global state σ that the internal execution halted in, the unspent gas *gas* from the internal transaction execution and the return data d.

The state of a non-terminated internal transaction is described by a regular execution state of the form (μ, ι, σ). The state is determined by the current global state σ of the system as well as the execution environment ι that specifies the parameters of the current transaction (including inputs and the code to be executed) and the local state μ of the stack machine.

Execution environment. The execution environment ι of an internal transaction is a tuple of static parameters $(\mathit{actor}, \mathit{input}, \mathit{sender}, \mathit{value}, \mathit{code})$ to the transaction that, i.a., determine the code to be executed and the account in whose context the code will be executed. The execution environment incorporates the following components: the active account *actor* that is the account that is currently executing and whose account will be affected when instructions for storage modification or money transfer are performed; the input data *input* given to the transaction; the address $sender$ of the account that initiated the transaction; the amount of wei *value* transferred with the transaction; the code *code* that is executed by the transaction. The execution environment is determined upon initialization of an internal transaction execution, and it can be accessed, but not altered during the execution.

Table 1: Semantic Rules for ADD

ADD

$$\frac{\iota.\mathit{code}\,[\mu.\mathsf{pc}] = \mathsf{ADD} \\ \mu.\mathsf{s} = a :: b :: s \qquad \mu.\mathsf{gas} \geq 3 \qquad \mu' = \mu[\mathsf{s} \rightarrow (a+b) :: s][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{-}= 3]}{\Gamma \vDash (\mu, \iota, \sigma) :: S \rightarrow (\mu', \iota, \sigma) :: S}$$

ADD-FAIL

$$\frac{\iota.\mathit{code}\,[\mu.\mathsf{pc}] = \mathsf{ADD} \qquad (|\mu.\mathsf{s}| < 2 \vee \mu.\mathsf{gas} < 3)}{\Gamma \vDash (\mu, \iota, \sigma) :: S \rightarrow \mathit{EXC} :: S}$$

Machine state. The local machine state μ represents the state of the underlying stack machine used for execution. Formally it is represented by a tuple (gas, pc, m, aw, s) holding the amount of gas *gas* available for execution, the program counter *pc*, the local memory *m*, the number of active words in memory *aw*, and the machine stack *s*.

The execution of each internal transaction starts in a fresh machine state, with an empty stack, memory initialized to all zeros, and program counter and active words in memory set to zero. Only the gas is instantiated with the gas value available for the execution. We call execution states with machine states of this form *initial*.

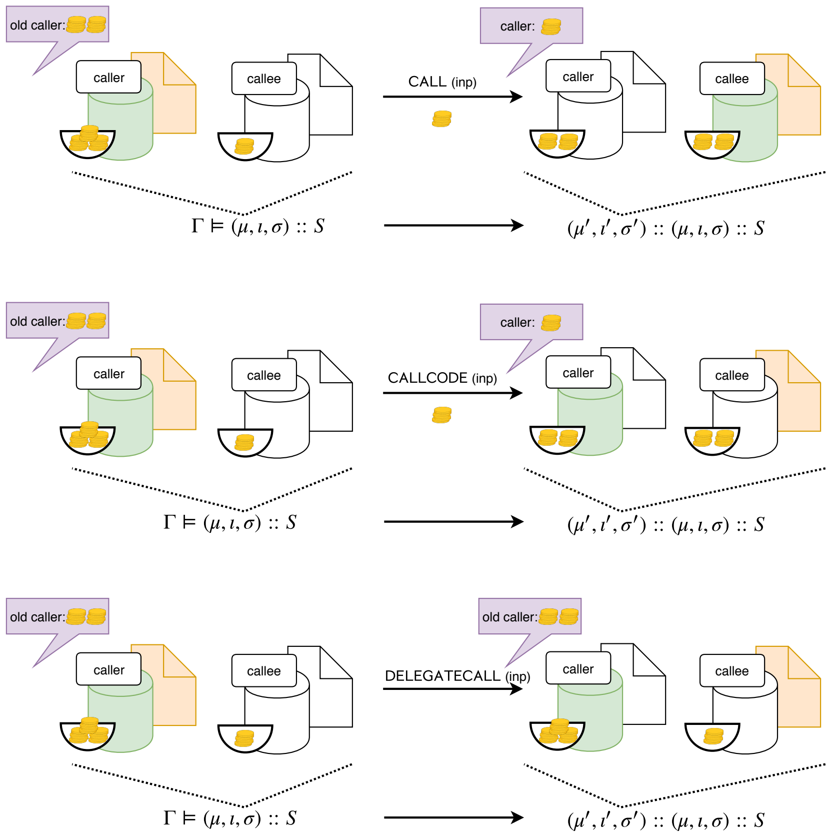
4.2 Small-step rules

In the following, we will present a selection of interesting small-step rules in order to illustrate the most important features of the semantics.

Local instructions. For demonstrating the overall design of the semantics, we start with the example of the arithmetic expression ADD performing addition of two values on the machine stack. The small-step rules for ADD are shown in Table 1. We use a dot notation, in order to access components of the different state parameters. We name the components with the variable names introduced for these components in the last section written in sans-serif-style. In addition, we use the usual notation for updating components: $t[\mathsf{c} \rightarrow v]$ denotes that the component c of tuple *t* is updated with value v. For expressing incremental updates in a simpler way, we additionally use the notation $t[\mathsf{c} \mathrel{+}= v]$ to denote that the (numerical) component of c is incremented by v and similarly $t[\mathsf{c} \mathrel{-}= v]$ for decrementing a component c of t.

The execution of the arithmetic instruction ADD only performs local changes in the machine state affecting the local stack, the program counter, and the gas budget. For deciding upon the correct instruction to execute, the currently executed code (that is part of the execution environment) is accessed at the position of the current program counter. The cost of an ADD instruction consists always of three units of gas that get subtracted from the gas budget in the machine state. As every other instruction, ADD can fail due to lacking gas or due to underflows on the machine stack. In this case, the exception state is entered and the execution of the current internal transaction is terminated. For better readability, we use here the slightly sloppy $\vee$ notation for combining the two error cases in one inference rule.

Transaction initiating instructions. A class of instructions with a more involved semantics are those instructions initiating internal transactions. This class incorporates instructions for calling another contract (CALL, CALLCODE and DELEGATECALL) and for creating a new contract (CREATE). We will explain the semantics of those instructions in an intuitive way omitting technical details.



old caller:
caller
callee
CALL (inp)
caller:
caller
callee
$\Gamma \vDash (\mu, \iota, \sigma) :: S$
$(\mu', \iota', \sigma') :: (\mu, \iota, \sigma) :: S$
old caller:
caller
callee
CALLCODE (inp)
caller:
caller
callee
$\Gamma \vDash (\mu, \iota, \sigma) :: S$
$(\mu', \iota', \sigma') :: (\mu, \iota, \sigma) :: S$
old caller:
caller
callee
DELEGATECALL (inp)
old caller:
caller
callee
$\Gamma \vDash (\mu, \iota, \sigma) :: S$
$(\mu', \iota', \sigma') :: (\mu, \iota, \sigma) :: S$

Fig. 1: Illustration of of the semantics of different call types

The call instructions initiate a new internal call transaction whose parameters are specified on the machine stack – including the recipient (callee) and the amount of money to be transferred (in the case of CALL and CALLCODE). In addition, the input to the call is specified by providing the corresponding local memory fragment and analogously a memory fragment for the return value.

When executing a call instruction, the specified amount of wei is transferred to the callee and the code of the callee is executed. The different call types diverge in the environment that the callee code is executed in. In the case of a CALL instruction, while executing the callee code (only) the account of the callee can be accessed and modified. So intuitively, the control is completely handed to the callee as its code is executed in its own context. In contrast, in the case of CALLCODE, the executed callee code can (only) access and modify the account of the caller. So the callee's code is executed in the caller's context which might be useful for using library functionalities implemented in a separate library contract that e.g., transfer money on behalf of the caller.

This idea is pushed even further in the DELEGATECALL instruction. This call type does not allow for transferring money and executes the callee's code not only in the caller's context, but even preserves part of the execution environment of the previous call (in particular the call value and the sender information). Intuitively, this instruction resembles adding the callee's code to the caller as an internal function so that calling it does not cause a new internal transaction (even though it formally does).

Figure 1 summarizes the behavior of the different call instructions in EVM bytecode. The executed code of the respective account is highlighted in orange while the accessible account state is depicted in green. The remaining internal transaction information (as specified in the execution environment) on the sender of the internal transaction and the transferred value are marked in violet. In addition, the picture relates the corresponding changes to the small-step semantics: the execution of a call transaction adds a new execution state to the call stack while preserving the old one. The new global state σ' records the changes in the accounts' balances, while the new execution environment ι' determines the accessible account (by setting the actor of the internal transaction correspondingly), the code to be executed (by setting code) and further accessible transaction information as the sender, value and input (by setting sender, value and input respectively).

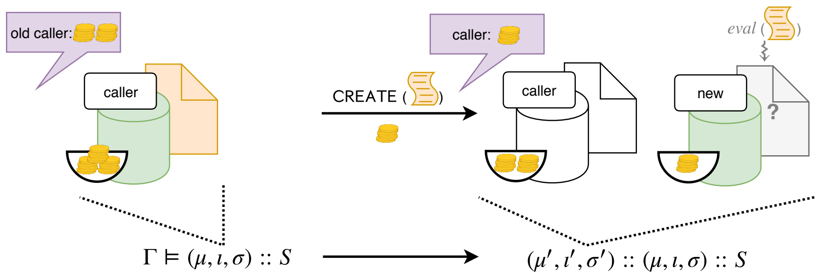

old caller:
caller
CREATE ( )
caller:
caller
eval ( )
new
?
$\Gamma \vDash (\mu, \iota, \sigma) :: S$
$(\mu', \iota', \sigma') :: (\mu, \iota, \sigma) :: S$

Fig. 2: Illustration of the semantics of the CREATE instruction

The CREATE instruction initiates an internal transaction that creates a new account. The semantics of this instruction is similar to the one of CALL, with the exception that a fresh account is created, which gets the specified value transferred, and that the input provided to this internal transaction, which is again specified in the local memory, is interpreted as the initialization code to be executed in order to produce the newly created account's code as output. Figure 2 depicts the semantics of the CREATE instruction in a similar fashion as it is done for the call instructions before. It is notable that the input to the CREATE instruction is interpreted as code and executed (therefore highlighted in orange) in the context of the newly created contract (highlighted in green). During this execution the newly created contract does not have any contract code itself (therefore depicted in gray), but only after completing the internal transaction the return value of the transaction will be set as code for the freshly created contract.

5 Security properties

Grishchenko et al. [15] propose generic security definitions for smart contracts that rule out certain classes of potentially harmful contract behavior. These properties constitute trace properties (more precisely, safety properties) as well as hyper properties (in particular, value independence properties). In this work, we revisit one of these safety properties called *single-entrancy* and use this property as a case study for showing how safety properties of smart contracts (that can be over-approximated by pure reachability properties) can be automatically checked by static analysis. For checking value independence properties, in [1] the reviewed analysis technique is extended with a simple dependency analysis that we will not discuss further in this work.

5.1 Preliminary Notations

Formally, contracts are represented as tuples of the form (a, code) where a denotes the address of the contract and *code* denotes the contract's code.

In order to give concise security definitions, we further introduce, and assume all through the paper, an annotation to the small step semantics in order to highlight the contract c that is currently executed. In the case of initialization code being executed, we use $\bot$. We write $S + +S'$ for the concatenation of call stacks S and S'. Finally, for arguing about EVM bytecode executions, we are only interested in those initial configurations that might result from a valid external transaction in a valid block. In the following, we will call these configurations *reachable* and refer to [15] for a detailed definition.

5.2 Single-entrancy

For motivating the definition of single-entrancy, we introduce a class of bugs in Ethereum smart contracts called *reentrancy bugs* [16,14].

The most famous representative of this class is the so-called DAO bug that led to a loss of 60 million dollars in June 2016 [11]. In an attack exploiting this bug, the affected contract was drained out of money by subsequently reentering it and performing transactions to the attacker on behalf of the contract.

The cause of such bugs mostly roots in the developer's misunderstanding of the semantics of Solidity's call primitives. In general, calling a contract can invoke two kinds of actions: Transferring Ether to the contract's account or Executing (parts of) a contracts code. In particular, Solidity's `call` construct (being translated to a CALL

```
contract Bob{
  bool sent = false;
  function ping( address c){
    if (!sent) { c.call.value(2)();
                 sent = true; }}}
```

(a) Smart contract with reentrancy bug

```
contract Mallory{
  function(){
    Bob(msg.sender).ping(this);}}
```

(b) Smart contract exploiting reentrancy bug

Fig. 3: Reentrancy Attack

instruction in EVM bytecode) invokes the execution of a fraction of the callee's code – specified in the so called *fallback function*. A contract's fallback function is written as a function without names or argument as depicted in the `Mallory` contract in Figure 3b.

Consequently, when using the `call` construct the developer may expect an atomic value transfer where potentially another contract's code is executed. For illustrating how to exploit this sort of bug, we consider the contracts in Figure 3.

The function `ping` of contract `Bob` sends an amount of 2 *wei* to the address specified in the argument. However, this should only be possible once, which is potentially ensured by the `sent` variable that is set after the successful money transfer. Instead, it turns out that invoking the `call.value` function on a contract's address invokes the contract's fallback function as well.

Given a second contract `Mallory`, it is possible to transfer more money than the intended 2 *wei* to the account of `Mallory`. By invoking `Bob`'s function `ping` with the address of `Mallory`'s account, 2 *wei* are transferred to `Mallory`'s account and additionally the fallback function of `Mallory` is invoked. As the fallback function again calls the `ping` function with `Mallory`'s address another 2 *wei* are transferred before the variable `sent` of contract `Bob` was set. This looping goes on until all gas of the initial call is consumed or the callstack limit is reached. In this case, only the last transfer of *wei* is reverted and the effects of all former calls stay in place. Consequently the intended restriction on contract `Bob`'s `ping` function (namely to only transfer 2 *wei* once) is circumvented.

Motivated by these kinds of attacks, the notion of single-entrancy was introduced. Intuitively, a contract is single-entrant if it cannot perform any more calls once it has been reentered. Formally this property can be expressed in terms of the small-steps semantics as follows:

Definition 1 (Single-entrancy [15]). *A contract c is single-entrant if for all reachable configurations $(\Gamma, s_c :: S)$, it holds for all s', s'', S' that*

$$\begin{aligned} &\Gamma \vDash s_c :: S \rightarrow^* {s'}_c :: S' ++ s_c :: S \\ \implies \neg\exists s'' \in \mathcal{S}, c' \in \mathcal{C}_\bot .\, &\Gamma \vDash {s'}_c :: S' ++ s_c :: S \rightarrow^* {s''}_{c'} :: {s'}_c :: S' ++ s_c :: S \end{aligned}$$

This property constitutes a safety property. We will show in § 7 how it can be appropriately abstracted for being expressed in the EtherTrust analysis framework.

$$\begin{tikzcd}
\Gamma, s_{c^*} :: S \arrow[d, Rightarrow] & & c^* \arrow[d, Rightarrow] & & \rightarrow^* & & S' ++ S \arrow[d, Rightarrow] \\
\Pi_s & \cup & \Delta & \vdash & \Pi & :> & \Pi_{S'}
\end{tikzcd}$$

Fig. 4: Simplified soundness statement

6 Verification

Grishchenko et al. [1] developed a static analysis framework for analyzing reachability properties of EVM smart contracts. This framwork relies on an abstract semantics for EVM bytecode soundly over-approximating the semantics presented in § 4.

In the following we will review the abstractions performed on the small-step configurations and execution rules using the example of the abstract execution rule for the ADD instruction. Afterwards, we will discuss shortly how call instructions are over-approximated.

6.1 Abstract semantics

Figure 4 gives an overview on the relation between the small-step and the abstract semantics. For the analysis, we will consider a particular contract c^* under analysis whose code is known. An over-approximation of the behavior of this smart contract will be encoded in *Horn clauses*($\varDelta$). These describe how an abstract configuration (represented by a set of abstract state predicates) evolves within the execution of the contract's instructions. Abstract configurations are obtained by translating small-step configurations to a set $\varPi$ of facts over state predicates that characterize (an over-approximation of) the original configuration. This transformation is performed with respect to the contract c^* as only all local behavior of this particular contract will be over-approximated and consequently only those elements on the callstack representing executions of c^* are translated. Finally, we will show that no matter how the contract c^* is called (so for every arbitrary reachable configuration $\varGamma, s_{c^*} :: S$), every sequence of execution steps that is performed while executing it can be mimicked by a derivation of the abstract configuration $\varPi_s$ (obtained from translating the execution state s) using the horn clauses $\varDelta$ (that model the abstract semantics of the contract c^*). More precisely, this means that from the set of facts $\varPi_s \cup \varDelta$ a set $\varPi$ can be derived that is a coarser abstraction ($<:$) than $\varPi_{S'}$ which is the translation of the execution's intermediate call stack S'. A corresponding formal soundness statement is proven in [1].

6.2 Abstract configurations

Table 2 shows the analysis facts used for describing the abstract semantics. These consist of (instances of) state predicates that represent partial abstract configurations. Accordingly, abstract configurations are sets of facts not containing any variables as arguments. We will refer to such facts as *closed facts*. Finally, abstract contracts are characterized as sets of Horn clauses over the state predicates (facts) that describe the state

Table 2: Analysis Facts. All arguments in the analysis facts marked with a hat $(\hat{\cdot})$ range over $\hat{D} \cup$ *Vars* where $\hat{D}$ is the abstract domain and *Vars* is the set of variables. All other arguments of analysis facts range over $\mathbb{N}$ with exception of *sa* that ranges over $(\mathbb{N} \rightarrow \hat{D}) \cup$ *Vars*. Closed facts *cf* are assumed to be facts with arguments not coming from *Vars*.

Facts	f	$:=$
Abs. machine state		$\mid$ $\mathsf{MState}_{pp}\,((size, sa), \hat{aw}, \hat{gas}, cd)$
Abs. memory		$\mid$ $\mathsf{Mem}_{pp}\,(\hat{pos}, \hat{va}, cd)$
Abs. exception state		$\mid$ $\mathsf{Exc}_{id^*}\,(cd)$
...		$\mid$ ...
Abs. configurations	$\varPi$	$:= \{cf_1, \ldots, cf_n\}$
Horn clauses	H	$:= \forall x^*. \bigwedge_i f_i \implies f$
Abs. contracts	$\varDelta$	$:= \{H_1, \ldots, H_n\}$

changes induced by the instructions at the different program positions. Here only those state predicates are depicted that are needed for describing the abstract semantics of the ADD instruction.

The state predicates are parametrized by a program point *pp* that is a tuple of the form (id^*, pc) with id^* being a contract identifier for contract c^* and *pc* being the program counter at which the abstract state holds.[3] The parametrization by the contract identifier helps to make the analysis consider a set of contracts whose code is known (such as e.g., library code that is known to be used by the contract). In this work however we focus on the case where c^* represented by identifier id^* is the only known contract. In addition, the predicates carry the relative call depth *cd* as argument. The relative call depth is the size of the call stack built up on the execution of c^* (Cf. call stack S' in Figure 4) and serves as abstraction for the (relative) call stack that contract c^* is currently executed on. The relative call depth helps to distinguish different recursive executions of c^* and thereby improves the precision of the analysis.

As the ADD instruction only operates on the local machine state, we focus on the abstract representation of the machine state μ: The state predicates representing μ are MState_{pp} and Mem_{pp}. The fact $\mathsf{MState}_{pp}\,((size, sa), \hat{aw}, \hat{gas}, cd)$ says that at program point *pp* and relative call depth *cd* the machine stack is of size *size* and its current configuration is described by the mapping *sa* which maps stack positions to abstract values, $\hat{aw}$ represents the number of active words in memory, and $\hat{gas}$ is the remaining gas. Similarly, the fact $\mathsf{Mem}_{pp}\,(\hat{pos}, \hat{v}, cd)$ states that at program point *pp* and relative call depth *cd* at memory address $\hat{pos}$ there is the (abstract) value $\hat{v}$. The values on the stack and in local memory range over an abstract domain. Concretely, we define the abstract domain $\hat{D}$ to be the set $\{\bot, \top, a^*\} \cup \mathbb{N}$ which constitutes a bounded lattice $(\hat{D}, \sqsubseteq, \sqcup, \sqcap, \top, \bot)$ satisfying $\bot \sqsubset a^* \sqsubset \top$ and $\bot \sqsubset n \sqsubset \top$ for all $n \in \mathbb{N}$. Intuitively, in our analysis $\top$ will represent unknown (symbolic) values and a^* will represent the unknown (symbolic) address of contract c^*.

[3] Making the program counter a parameter instead of an argument is a design choice made in order to minimize the number of recursive horn clauses simplifying automated verification.

Treating the address of the contract under analysis in a symbolic fashion is crucial for obtaining a meaningful analysis, as the address of this account on the blockchain can not easily be assumed to be known upfront. Although discussing this peculiarity is beyond the scope of this paper, a broader presentation of the symbolic address paradigm can be found in the technical report [1].

For performing operations and comparisons on values from the abstract domain, we will assume versions of the unary, binary and comparison operators on the values from $\hat{D}$. We will mark abstract operators with a hat ($\hat{\cdot}$) and e.g., write $\widehat{+}$ for abstract addition or $\widehat{=}$ for abstract equality. The operators will treat $\top$ and a^* as arbitrary values so that e.g., $\top \mathbin{\widehat{+}} n$ evaluates to $\top$ and $\top \mathbin{\widehat{=}} n$ evaluates to *true* and *false* for all $n \in \mathbb{N}$.

Formally, we establish the relation between a concrete machine state μ and its abstraction by an abstraction function that translates machine states to a set of closed analysis facts. Figure 3 shows the abstraction function α_μ that maps a local machine state into an abstract state consisting of a set of analysis facts. The abstraction is defined with respect to the relative call depth *cd* of the execution and a value abstraction function $\mathring{\cdot}$ that maps concrete values into values from the abstract domain. The function $\mathring{\cdot}$ thereby maps all concrete values to the corresponding (concrete) values in the abstract domain, but those values that can potentially represent the address of contract c^*, hence, they are translated to a^* and therefore over-approximated. This treatment might introduce spurious counterexamples with respect to the concrete execution of the real contract on the blockchain (where it is assigned a concrete address). On the one hand, this is due to the fact that by this abstraction the concrete value of the address is assumed to be arbitrary. On the other hand, abstract computations with α always result in $\top$ and therefore possible constraints on these results are lost. However, the first source of imprecision should not be considered an imprecision per se, as the c^*'s address is not assumed to be known statically, thus, the goal of the abstraction is to over-approximate the executions with all possible addresses.

The translation proceeds by creating a set of instances of the machine state predicates. For creating instances of the MState_{pp} predicate, the concrete values *aw* and *gas* are over-approximated by $\mathring{aw}$ and $\mathring{gas}$ respectively, and the stack is translated to an abstract array representation using the function $\mathsf{stackToArray}$. The instances of the memory predicate are created by translating the memory mapping m to a relational representation with abstract locations and values. [4]

6.3 Abstract execution rules

As all state predicates are parametrized by their program points, the abstract semantics needs to be formulated with respect to program points as well. More precisely this means that for each program counter of contract c^* a set of Horn clauses is created that describes the semantics of the instruction at this program counter. Formally, a function

[4] The reason for using a separate predicate for representing local memory instead of encoding it as an argument of array type in the main machine state predicate is purely technical: for modeling memory usage correctly we would need a rich set of array operations that are however not supported by the fixedpoint engines of modern SMT solvers.

Table 3: Abstraction function for the local machine state μ

$$\alpha_\mu\left((gas, pc, m, aw, s), cd\right) := \{\mathsf{MState}_{(id^*,\, pc)}\left(\mathsf{stackToArray}\left(s\right), \mathring{aw}, \mathring{gas}, cd\right)\}$$
$$\cup \{\mathsf{Mem}_{(id^*,\, pc)}\left(\mathring{pos}, \mathring{v}, cd\right) \mid m\left[pos\right] = v \wedge pos \leq 2^{256}\}$$

$$\mathsf{stackToArray}\left(\epsilon\right) := (0, \lambda x.\, 0)$$
$$\mathsf{stackToArray}\left(x :: s\right) := let\ (size, sa) = \mathsf{stackToArray}\left(s\right)\ in\ (size + 1, sa_{\mathring{x}}^{size})$$

$(\!|\cdot|\!)_{pp}^{\{c^*\}}$ is defined that creates the required set of rules given that the instruction *inst* is at position *pc* of contract c^*'s code.

Table 4 shows parts of the the definition of $(\!|\cdot|\!)_{pp}^{\{c^*\}}$ for the ADD instruction. The main functionality of the rule is described by the Horn clause 1 that describes how the machine stack and the gas evolve when executing ADD. First the precondition is checked whether the sufficient amount of gas and stack elements are available. Then the two (abstract) top elements $\hat{x}$ and $\hat{y}$ are extracted from the stack and their sum is written to the top of the stack while reducing the overall stack size by 1. In addition, the local gas value is reduced by 3 in an abstract fashion. In the memory rule (Horn clause 2), again the preconditions are checked and then (as memory is not affected by the ADD instruction) the memory is propagated. This propagation is needed due to the memory predicate's parametrization with the program counter: For making the memory accessible in the next execution step, its values need to be written into the corresponding predicate for the next program counter. Finally, Horn clauses 3 and 4 characterize the exception cases: an exception while executing the ADD instruction can occur either because of a stack underflow or as the execution runs out of gas. In both cases the exception state is entered which is indicated by recording the relative call depth of the exception in the predicate $\mathsf{Exc}_{id^*}\left(cd\right)$.

By allowing gas values to come from the abstract domain, we enable symbolic treatment of gas. In particular this means that when starting the analysis with gas value $\top$, all gas calculations will directly result in $\top$ again (and could therefore be omitted) and in particular all checks on the gas will result in *true* and *false* and consequently always both paths (regular execution via Horn clauses 1 and 2 and exception via Horn clause 4) will be triggered in the analysis.

For over-approximating the semantics of call instructions, more involved abstractions are needed. We will illustrate these abstractions in the following in an intuitive way and refer to [1] for the technical details. Note that in the following we will assume CALL instructions to be the only kind of transaction initiating instructions that are contained in the contracts that we consider for analysis. A generalization of the analysis that allows for incorporating also other call types is presented in [1].

As we are considering c^* the only contract to be known, whenever a call is performed that is not a self-call, we need to assume that an arbitrary contract $c^?$ gets ex-

Table 4: Exerpt of the abstract rules for ADD

$$
\begin{aligned}
(\!|\mathsf{ADD}|\!)^{\{c^*\}}_{(id*,pc)} = \{ & \mathsf{MState}_{(id^*,\,pc)}\,((size, sa), \hat{aw}, \hat{gas}, cd) \land size > 1 \land \hat{gas} \mathrel{\widehat{\geq}} 3 \\
& \land \hat{x} = sa[size-1] \land \hat{y} = sa[size-2] \\
& \Rightarrow \mathsf{MState}_{(id^*,\,pc+1)}\,((size-1, sa^{size-2}_{\hat{x} \mathbin{\widehat{+}} \hat{y}}), \hat{aw}, \hat{gas} \mathbin{\widehat{-}} 3, cd), & (1) \\
& \mathsf{Mem}_{(id^*,\,pc)}\,(\hat{pos}, \hat{va}, cd) \land \mathsf{MState}_{(id^*,\,pc)}\,((size, sa), \hat{gas}, \hat{aw}, cd) \\
& \land size > 1 \land \hat{gas} \mathrel{\widehat{\geq}} 3 \Rightarrow \mathsf{Mem}_{(id^*,\,pc+1)}\,(\hat{pos}, \hat{va}, cd), & (2) \\
& \mathsf{MState}_{(id^*,\,pc)}\,((size, sa), \hat{gas}, \hat{aw}, cd) \land size < 2 \Rightarrow \mathsf{Exc}_{id*}\,(cd), & (3) \\
& \mathsf{MState}_{(id^*,\,pc)}\,((size, sa), \hat{gas}, \hat{aw}, cd) \land \hat{gas} \mathrel{\widehat{<}} 3 \Rightarrow \mathsf{Exc}_{id*}\,(cd) \ldots \} & (4)
\end{aligned}
$$

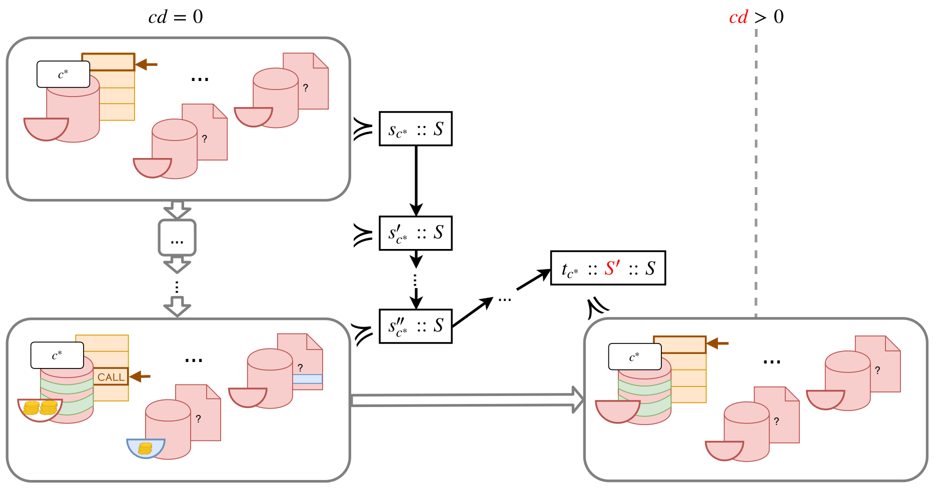


Fig. 5: Illustration of the abstraction of the semantics for the CALL instruction.

ecuted. The general idea for over-approximating calls to an unknown contract $c^?$ is that only those execution states that represent executions of contract c^* will be over-approximated. Consequently, when a call is performed, all possible effects on future executions of c^* that might be caused by the execution of $c^?$ (including the initiation of further initial transactions that might cause reentering c^*) need to be captured. For doing this as accurate as possible, we use the following observations:

1. Given that c^* only executes plain CALL instructions the persistent storage of contract c^* can only be altered during executions of c^*.
2. Contracts have a single entry point: their execution always starts in a fresh machine state at program counter zero.

In general, we can soundly capture the possibility of contract c^* being reentered during the execution of $c^?$ by assuming to reenter c^* at every higher call level. For keeping the desired precision, we can use the previously made observations for imposing restrictions on the reenterings of c^*: First, we assume the persistent storage of c^* to be the same as at the point of calling (observation 1.). Second, we know that execution starts at program counter 0 in a fresh machine state (observation 2.). This allows us to initialize the machine state predicates presented in Table 2 accordingly at program counter zero. All other parts of the global state and the execution environment need to be considered unknown at the point of reentering as they might have potentially been changed during the execution of $c^?$. This in particular also applies to the balance of contract c^*.

Figure 5 illustrates how the abstract configurations over-approximating the concrete execution states of c^* evolve within the execution of the abstract semantics. We write $\Pi \succcurlyeq S$ for denoting that an abstract configuration Π (here graphically depicted in gray frames) is an over-approximation of call stack S. The depicted execution starts in the initial execution state s_{c^*} of c^*. This is state is over-approximated by assuming the storage and balance of c^* as well as all other information on the global state to be unknown and therefore initialized to $\top$ in the corresponding state predicates of the abstract configuration (denoted in the picture by marking the corresponding state components in red). The execution steps representing the executions of local instructions are mimicked step-wise by corresponding abstract execution steps. During these steps a more refined knowledge about the state of c^* and its environment might be gained (e.g., the value of some storage cells where information is written, or some restrictions on the account's balances, marked in green or blue, respectively). When finally a CALL instruction is executed, every potential reentering of contract c^* (here exemplified by execution state t_{c^*}) is over-approximated by abstract configurations for every call depths $cd > 0$ that consider all global state and environmental information to be arbitrary, but the parts modeling the persistent storage of c^* to be as at the point of calling. In § 7 we will show how this abstraction will help us to automatically check smart contracts for single-entrancy in a sound and precise manner. In addition to these over-approximations that capture the effects on c^* during the execution of an unknown contract, for over-approximating CALL instructions some other abstractions need to be performed that model the semantics of returning:

- For returning it is always assumed that potentially the call failed or returned with arbitrary return values.
- After returning the global state is assumed to be altered arbitrarily by the call and therefore its components are set to $\top$.

For a complete account and formal description of the abstractions, we refer to the full specification of the abstract semantics spelled out in the technical report [1].

7 Verifying security properties

In this section, we will show how the previously presented analysis can be used for proving reachability properties of Ethereum smart contracts in an automated fashion.

To this end, we review EtherTrust [1], the first sound static analyzer for EVM bytecode. EtherTrust proceeds by translating contract code provided in the bytecode format into an internal Horn clause representation. This Horn clause representation, together with facts over-approximating all potential initial configurations are handed to the SMT solver Z3 [45] via an API. For showing that the analyzed contract satisfies a reachability property, the unsatisfiability of the corresponding analysis queries needs to be verified using Z3's fixedpoint engine SPACER [46]. If all analysis queries are deemed unsatisfiable then the contract under analysis is guaranteed to satisfy the original reachability query due to the soundness of the underlying analysis.

In the following we will discuss the analysis queries used for verifying single-entrancy and illustrate how these queries allow for capturing contracts that are vulnerable to reentrancy such as the example presented in § 5.

7.1 Over-approximating Single-entrancy

For being able to automatically check for single-entrancy, we need to simplify the original property in order to obtain a description that is expressible in terms of the analysis framework described in § 6. To this end, a strictly stronger property named *call unreachability* is presented that is proven to imply single-entrancy:

Definition 2 (Call unreachability [1]). *A contract c is call unreachable if for all initial execution states (μ, ι, σ) such that $(\mu, \iota, \sigma)_c$ is well formed, it holds that for all transaction environments Γ and all call stacks S*

$$\begin{aligned} \neg \exists s, S'.\, \Gamma \vDash (\mu, \iota, \sigma)_c :: S \rightarrow^* s_c :: S' + +S \\ \wedge\ |S'| > 0\ \wedge\ \mathit{code}\,(c)\,[s.\mu.\mathsf{pc}] \in \mathit{Inst}_{\mathit{call}} \end{aligned}$$

With $\mathit{Inst}_{\mathit{call}} = \{\mathsf{CALL}, \mathsf{CALLCODE}, \mathsf{DELEGATECALL}, \mathsf{CREATE}\}$

Intuitively, this property states that it should not be possible to reach a call instruction of c^* after reentering. As we are excluding all transaction initiating instructions but CALL from the analysis, it is sufficient to query for the reachability of a CALL instruction of c^* on a higher call depth. More precisely, we end up with the following set of queries:

$$\{\mathsf{MState}_{(\mathit{id},\,\mathsf{pc})}\,((\mathit{size}, \mathit{sa}), \mathit{aw}, \mathit{gas}, \mathit{cd}) \wedge \mathit{cd} > 0 \mid \mathit{code}\,(c^*)\,[\mathit{pc}] = \mathsf{CALL}\} \qquad (5)$$

As the MState_{pp} predicate tracks the state of the machine state at all program points, it can be used as indicator for reachability of the program point as such. Consequently, by querying the $\mathsf{MState}_{(\mathit{id}^*,\,\mathsf{pc})}$ for all program counters pc where c^* has a CALL instruction and along with that requiring a call depth exceeding zero, we can check whether a call instruction is reachable in some reentering execution.

7.2 Examples

We will use examples for showing how the analysis detects, and proves the absence of reentrancy bugs, respectively. To this end, we revisit the contract `Bob` presented in § 5, and introduce a contract `Alice` that fixes the reentrancy bug that is present in `Bob`. The two contracts are shown in Figure 6.

Detecting reentrancy bugs. We illustrate how the analysis detects reentrancy bugs using the example in Figure 6a. To this end we give a graphical description of the over-approximations performed when analyzing contract `Bob` which is depicted in Figure 7.

```
1 contract Bob{
2   bool sent = false;
3   function ping( address c){
4     if (!sent) { c.call.value(2)();
5                  sent = true; }}}
```

(a) Smart contract with reentrancy bug

```
1 contract Alice{
2   bool sent = false;
3   function ping( address c){
4     if (!sent) { sent = true;
5                  c.call.value(2)(); }}}
```

(b) Smart contract with fixed reentrancy bug

Fig. 6: Examples for contracts showing and being robust against the reentrancy bug.

For the sake of presentation, we give the contract code in Solidity instead of bytecode and argue about it on this level even though the analysis is carried out on bytecode level.

As discussed in § 6.3, the analysis considers the execution of contract `Bob` to start in an unknown environment, which implies that also the value of the contract's `sent` variable is unknown and hence initialized to $\top$. As a consequence, the equality check in line 4 is considered to evaluate to both *true* and *false* in the abstract setting (as $\top$ needs to be considered to potentially equal every concrete value). Accordingly, the analysis needs to consider the `then`-branch of the conditional and consequently the call in line 4. This call is over-approximated as discussed in § 6.3, and therefore considers reentering contract `Bob` in an arbitrary call depth. In this situation, the `sent` variable is still over-approximated to have value $\top$ wherefore the call at line 4 can be reached again which satisfies the reachability query in Equation 5.

Proving single-entrancy. We consider the contract `Alice` shown in Figure 6b. In contrast to contract `Bob`, this contract does not have the reentrancy vulnerability, as the guard `sent` that should prevent the call instruction in line 5 from being executed more than once is set before performing the call. As a consequence, when reentering the contract, the guard is already set and stops any further calls. We show that the analysis presented in § 6 is precise enough for proving this contract to be single-entrant. Intuitively, the abstraction is precise as it considers that the contract's persistent storage can be assumed to be unchanged at the point of reentering. Consequently, the `then`-branch of the conditional can be excluded from the analysis when reentering and the contract can be proven to be single-entrant. A graphic description of this argument is provided in Figure 8. As for contract `Bob`, the analysis starts in an abstract configuration that assigns the `sent` variable value $\top$, which forces the analysis to consider the `then` as well as the `else`-branch of the conditional in line 4. When taking the `else`-branch, the contract execution terminates without reaching a state satisfying the reachability query. Therefore, it is sufficient to only consider the `then`-branch for proving the impossibility of re-reaching the call instruction. When executing the `call` in the `then`-branch, according to the abstract call semantics, the analysis needs to take all abstract configurations representing executions of `Alice` at higher call depths into account. However, in each of these abstract configurations it can be assumed that the state of the persistent storage (including the `sent` variable, highlighted in green) is the same as at the point of calling. As at this point `sent` was already initialized to the concrete value **`true`**, the `then`-branch of the conditional can be excluded from the analysis at any call depth $cd > 0$ and consequently the unreachability of the query in Equation 5 is proven.

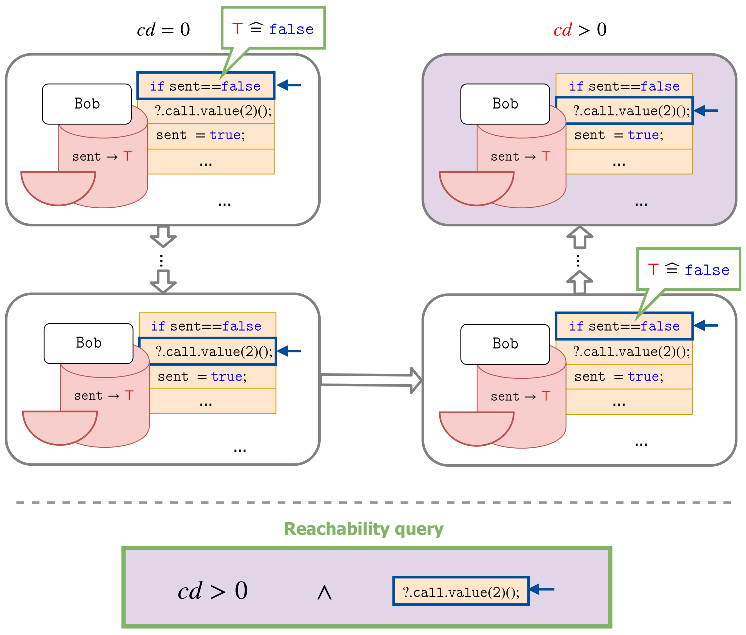


Fig. 7: Illustration of the attack detection in contract `Bob` by the static analysis.

7.3 Discussion

In this section, we illustrated how the static analysis underlying EtherTrust [1] in principle is capable not only of detecting re-entrancy bugs, but also of proving smart contracts single-entrant. In practice, EtherTrust manages to analyze real-world contracts from the blockchain within several seconds, as detailed in the experimental evaluation presented in [1]. Even though EtherTrust produces false positives due to the performed over-approximations, it still shows better precision on a benchmark than the state-of-the art bug-finding tool Oyente [16] – despite being sound. Similar results are shown when using EtherTrust for checking a simple value independency property.

In general, EtherTrust could be easily extended to support more properties on contract execution – given that those properties or over-approximations of them are expressible as reachability or simple value independency properties. By contrast, checking more involved hyper properties, or properties that span more than one external transaction execution is currently out of the scope for EtherTrust.

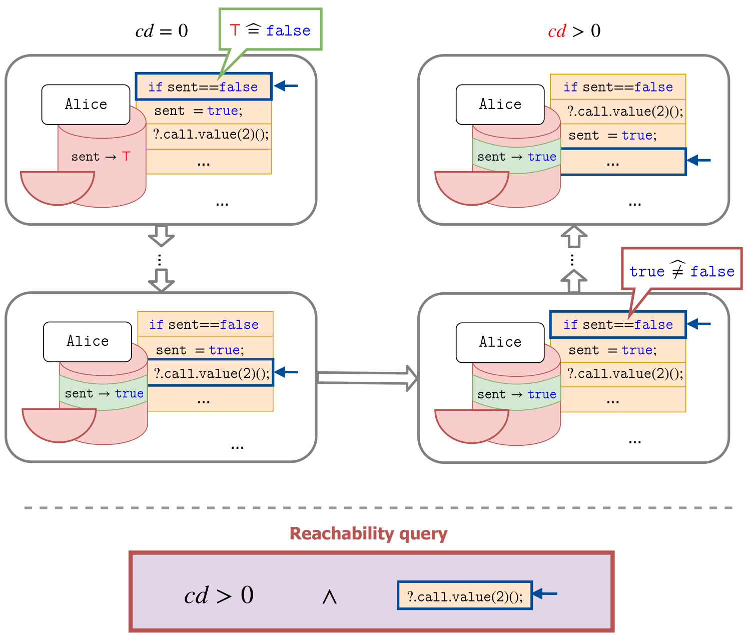


Fig. 8: Illustration of of proving single-entrancy of contract `Alice` by the static analysis.

8 Conclusion

We presented a systematization of the state-of-the-art in Ethereum smart contract verification and outlined the open challenges in this field. Also we discussed in detail the foundations of EtherTrust [1], the first sound static analyzer for EVM bytecode. In particular, we reviewed how the small-step semantics presented in [15] is abstracted into a set of Horn clauses. Also we presented how single-entrancy – a relevant smart contract security property – is expressed in terms of queries, which can be then automatically solved leveraging the power of an SMT solver.

References

1. : Ethertrust: Technical report Available at `https://www.netidee.at/ethertrust`.
2. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system (2008) Available at `http://bitcoin.org/bitcoin.pdf`.
3. Hahn, A., Singh, R., Liu, C.C., Chen, S.: Smart contract-based campus demonstration of decentralized transactive energy auctions. In: Power & Energy Society Innovative Smart Grid Technologies Conference (ISGT), 2017 IEEE, IEEE (2017) 1–5
4. Adhikari, C.: Secure framework for healthcare data management using ethereum-based blockchain technology. (2017)

5. Biryukov, A., Khovratovich, D., Tikhomirov, S.: Findel: Secure derivative contracts for ethereum. In: International Conference on Financial Cryptography and Data Security, Springer (2017) 453–467
6. McCorry, P., F. Shahandashti, S., Hao, F.: A smart contract for boardroom voting with maximum voter privacy. Proceedings of the Financial Cryptography and Data Security Conference (2017)
7. Notheisen, B., Gödde, M., Weinhardt, C.: Trading stocks on blocks-engineering decentralized markets. In: International Conference on Design Science Research in Information Systems, Springer (2017) 474–478
8. Mathieu, F., Mathee, R.: Blocktix: Decentralized event hosting and ticket distribution network. (2017) Available at `https://blocktix.io/public/doc/blocktix-wp-draft.pdf`.
9. Azaria, A., Ekblaw, A., Vieira, T., Lippman, A.: Medrec: Using blockchain for medical data access and permission management. In: Open and Big Data (OBD), International Conference on, IEEE (2016) 25–30
10. Dong, C., Wang, Y., Aldweesh, A., McCorry, P., van Moorsel, A.: Betrayal, distrust, and rationality: Smart counter-collusion contracts for verifiable cloud computing. (2017)
11. : The DAO smart contract (2016) Available at `http://etherscan.io/address/0xbb9bc244d798123fde783fcc1c72d3bb8c189413#code`.
12. : The parity wallet breach (2017) Available at `https://www.coindesk.com/30-million-ether-reported-stolen-parity-wallet-breach/`.
13. : The parity wallet vulnerability (2017) Available at `https://paritytech.io/blog/security-alert.html`.
14. Atzei, N., Bartoletti, M., Cimoli, T.: A survey of attacks on ethereum smart contracts (sok). In: International Conference on Principles of Security and Trust, Springer (2017) 164–186
15. Grishchenko, I., Maffei, M., Schneidewind, C.: A semantic framework for the security analysis of ethereum smart contracts. In: Proceedings of the 7th International Conference on Principles of Security and Trust (POST), Springer (2018)
16. Luu, L., Chu, D.H., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, ACM (2016) 254–269
17. Zhou, E., Hua, S., Pi, B., Sun, J., Nomura, Y., Yamashita, K., Kurihara, H.: Security assurance for smart contract. In: New Technologies, Mobility and Security (NTMS), 2018 9th IFIP International Conference on, IEEE (2018) 1–5
18. Nikolic, I., Kolluri, A., Sergey, I., Saxena, P., Hobor, A.: Finding the greedy, prodigal, and suicidal contracts at scale. arXiv preprint arXiv:1802.06038 (2018)
19. Kalra, S., Goel, S., Dhawan, M., Sharma, S.: Zeus: Analyzing safety of smart contracts, NDSS (2018)
20. Buenzli, F., Dan, A., Drachsler-Cohen, D., Gervais, A., Tsankov, P., Vechev, M.: Securify (2017) Available at `http://securify.ch`.
21. : Mythril Available at `https://github.com/ConsenSys/mythril`.
22. : Manticore Available at `https://github.com/trailofbits/manticore`.
23. SmartDec: Smartcheck. `https://github.com/smartdec/smartcheck`
24. : Solgraph Available at `https://github.com/raineorshine/solgraph`.
25. Hirai, Y.: Defining the ethereum virtual machine for interactive theorem provers. In: International Conference on Financial Cryptography and Data Security, Springer (2017) 520–535
26. Wood, G.: Ethereum: A secure decentralised generalised transaction ledger. Ethereum Project Yellow Paper **151** (2014) 1–32
27. Amani, S., Bégel, M., Bortin, M., Staples, M.: Towards verifying ethereum smart contract bytecode in isabelle/hol. CPP. ACM. To appear (2018)

28. Hildenbrandt, E., Saxena, M., Zhu, X., Rodrigues, N., Daian, P., Guth, D., Rosu, G.: Kevm: A complete semantics of the ethereum virtual machine. Technical report (2017)
29. Roşu, G., Şerbănută, T.F.: An overview of the k semantic framework. The Journal of Logic and Algebraic Programming **79**(6) (2010) 397–434
30. Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Gollamudi, A., Gonthier, G., Kobeissi, N., Kulatova, N., Rastogi, A., Sibut-Pinote, T., Swamy, N., et al.: Formal verification of smart contracts: Short paper. In: Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security, ACM (2016) 91–96
31. Grossman, S., Abraham, I., Golan-Gueta, G., Michalevsky, Y., Rinetzky, N., Sagiv, M., Zohar, Y.: Online detection of effectively callback free objects with applications to smart contracts. Proceedings of the ACM on Programming Languages **2**(POPL) (2017) 48
32. Cook, T., Latham, A., Lee, J.H.: Dappguard: Active monitoring and defense for solidity smart contracts
33. OConnor, R.: Simplicity: A new language for blockchains. arXiv preprint arXiv:1711.03028 (2017)
34. Pettersson, J., Edström, R.: Safer smart contracts through type-driven development
35. Coblenz, M.: Obsidian: A safer blockchain programming language. In: Software Engineering Companion (ICSE-C), 2017 IEEE/ACM 39th International Conference on, IEEE (2017) 97–99
36. Schrans, F., Eisenbach, S., Drossopoulou, S.: Writing safe smart contracts in flint
37. : Vyper Available at `https://github.com/ethereum/vyper`.
38. : Bamboo Available at `https://github.com/pirapira/bamboo`.
39. : Formal verification for solidity contracts. available at `https://forum.ethereum.org/discussion/3779/formal-verification-for-solidity-contracts`
40. Filliâtre, J.C., Paskevich, A.: Why3where programs meet provers. In: European Symposium on Programming, Springer (2013) 125–128
41. Sergey, I., Kumar, A., Hobor, A.: Scilla: a smart contract intermediate-level language. arXiv preprint arXiv:1801.00687 (2018)
42. Wöhrer, M., Zdun, U.: Smart contracts: Security patterns in the ethereum ecosystem and solidity. (2018)
43. Mavridou, A., Laszka, A.: Designing secure ethereum smart contracts: A finite state machine based approach. arXiv preprint arXiv:1711.09327 (2017)
44. Sergey, I., Hobor, A.: A concurrent perspective on smart contracts. arXiv preprint arXiv:1702.05511 (2017)
45. De Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: International conference on Tools and Algorithms for the Construction and Analysis of Systems, Springer (2008) 337–340
46. Komuravelli, A., Gurfinkel, A., Chaki, S.: Smt-based model checking for recursive programs. Form. Methods Syst. Des. **48**(3) (June 2016) 175–205