A Semantic Framework for the Security Analysis of Ethereum Smart Contracts

Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind(✉)

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

Abstract. Smart contracts are programs running on cryptocurrency (e.g., Ethereum) blockchains, whose popularity stem from the possibility to perform financial transactions, such as payments and auctions, in a distributed environment without need for any trusted third party. Given their financial nature, bugs or vulnerabilities in these programs may lead to catastrophic consequences, as witnessed by recent attacks. Unfortunately, programming smart contracts is a delicate task that requires strong expertise: Ethereum smart contracts are written in Solidity, a dedicated language resembling JavaScript, and shipped over the blockchain in the EVM bytecode format. In order to rigorously verify the security of smart contracts, it is of paramount importance to formalize their semantics as well as the security properties of interest, in particular at the level of the bytecode being executed.

In this paper, we present the first complete small-step semantics of EVM bytecode, which we formalize in the F* proof assistant, obtaining executable code that we successfully validate against the official Ethereum test suite. Furthermore, we formally define for the first time a number of central security properties for smart contracts, such as call integrity, atomicity, and independence from miner controlled parameters. This formalization relies on a combination of hyper- and safety properties. Along this work, we identified various mistakes and imprecisions in existing semantics and verification tools for Ethereum smart contracts, thereby demonstrating once more the importance of rigorous semantic foundations for the design of security verification techniques.

1 Introduction

One of the determining factors for the growing interest in blockchain technologies is the groundbreaking promise of secure distributed computations even in absence of trusted third parties. Building on a distributed ledger that keeps track of previous transactions and the state of each account, whose functionality and security is ensured by a delicate combination of incentives and cryptography, software developers can implement sophisticated distributed, transactions-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 [1]), Ethereum was designed from the

L. Bauer and R. Küsters (Eds.): POST 2018, LNCS 10804, pp. 243–269, 2018.
https://doi.org/10.1007/978-3-319-89722-6_10

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 financial contracts [2], auctions [3], elections [4], data management systems [5], trading platforms [6,7], permission management [8] and verifiable cloud computing [9], just to mention a few. Given their financial nature, bugs and vulnerabilities in smart contracts may lead to catastrophic consequences. For instance, the infamous DAO vulnerability [10] recently led to a 60M$ financial loss and similar vulnerabilities occur on a regular basis [11,12]. Furthermore, many smart contracts in the wild are intentionally fraudulent, as highlighted in a recent survey [13].

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 a 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.

Related Work. Recognizing the importance of solid semantic foundations for smart contracts, the Ethereum foundation published a yellow paper [14] to describe the intended behaviour of smart contracts. This semantics, however, exhibits several under-specifications and does not follow any standard approach for the specification of program semantics, thereby hindering program verification. In order to provide a more precise characterization, Hirai formalizes the EVM semantics in the proof assistant Isabelle/HOL and uses it for manually proving safety properties for concrete programs [15]. This semantics, however, constitutes just a sound over-approximation of the original semantics [14]. More specifically, once a contract performs a call that is not a self-call, it is assumed that arbitrary code gets executed and consequently arbitrary changes to the account's state and to the global state can be performed. Consequently, this semantics can not serve as a general-purpose basis for static analysis techniques that might not rely on the same over-approximation.

In a concurrent, unpublished work, Hildebrandt et al. [16] define the EVM semantics in the $\mathbb{K}$ framework [17] – 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 underlying semantics relies on non-standard local rewriting rules on the system configuration. Since parts of the execution are treated in separation such as the exception behavior and the gas calculations, one small-step consists of several rewriting steps, which makes

[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.

this semantics harder to use as a basis for new static analysis techniques. This is relevant whenever the static analysis tools derivable by the $\mathbb{K}$ framework are not sufficient for the desired purposes: for instance, their analysis requires the user to manually specify loop invariants, which is hardly doable for EVM bytecode and clearly does not scale to large programs. Furthermore, all these works concentrate on the semantics of EVM bytecode but do not study security properties for smart contracts.

Sergey and Hobor [18] compare smart contracts on the blockchain with concurrent objects using shared memory and use this analogy to explain typical problems that arise when programming smart contracts in terms of concepts known from concurrency theory. They encourage the application of state-of-the art verification techniques for concurrent programs to smart contracts, but do not describe any specific analysis method applied to smart contracts themselves. Mavridou and Laszka [19] define a high-level semantics for smart contracts that is based on finite state machines and aims at simplifying the development of smart contracts. They provide a translation of their state machine specification language to Solidity, a higher-order language for writing Ethereum smart contracts, and present design patterns that should help users to improve the security of their contracts. The translation to Solidity is not backed up by a correctness proof and the design patterns are not claimed to provide any security guarantees.

Bhargavan et al. [20] 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.

Luu et al. have recently presented Oyente [21], a state-of-the-art static analysis tool for EVM bytecode that relies on symbolic execution. Oyente comes with a semantics of a simplified fragment of the EVM bytecode and, in particular, misses several important commands related to contract calls and contract creation. Furthermore, it is affected by a major bug related to calls as well as several other minor ones which we discovered while formalizing our semantics, which is inspired by theirs. Oyente supports a variety of security properties, such as transaction order dependency, timestamp dependency, and reentrancy, but the security definitions are rather syntactic and described informally. As we show in this paper, the lack of solid semantic foundations causes several sources of unsoundness in Oyente.

Our Contributions. This work lays the semantic foundations for Ethereum smart contracts. Specifically, we introduce

- The first complete small-step semantics for EVM bytecode;
- A formalization in F* of a large fragment of our semantics, which can serve as a foundation for verification techniques based on encoding into this language [20] as well as machine-checked proofs for other analysis techniques (e.g., [21]). By compiling F* in OCaml, we could successfully validate our semantics against the official Ethereum test suite;

- The first formal definitions of crucial security properties for smart contracts, such as call integrity, for which we devise a dedicated proof technique, atomicity, and independence from miner controlled parameters. Interestingly enough, the formalization of these properties requires hyper-properties, while existing static analysis techniques for smart contracts rely on reachability properties and syntactic conditions;
- A collection of examples showing how the syntactic conditions employed in current analysis techniques are imprecise and, in several cases, unsound, thereby further motivating the need for solid semantic foundations and rigorous security definitions for smart contracts.

The complete semantics as well as the formalization in F* are publicly available [22].

Outline. The remainder of this paper is organized as follows. Section 2 briefly overviews the Ethereum architecture, Sect. 3 introduces the Ethereum semantics and our formalization in F*, Sect. 4 formally defines various security properties for Ethereum smart contracts, and Sect. 5 concludes highlighting interesting research directions.

2 Background on Ethereum

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 prize

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

(the amount of wei to pay for a unit of gas). Upfront, the originator pays for the gas limit according to the gas prize 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.

EVM Bytecode. The code of contracts is written in *EVM bytecode* – 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.

Gas and Exceptions. The execution of each instruction consumes a positive amount of gas. There is a gas limit set by the sender of the transaction. Exceeding the gas limit 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.

Solidity. In practice, most Ethereum smart contracts are not written in EVM bytecode directly, but in the high-level language Solidity which is developed by the Ethereum Foundation [23]. For understanding the typical problems that arise when writing smart contracts, it is important to consider the design of this high-level language.

Solidity is a so called "contract-oriented" programming language that uses the concept of class from object-oriented languages for the representation of contracts. Similar to classes in object-oriented programming, contracts specify fields and methods for contract instances. Fields can be seen as persistent storage of a contract (instance) and contract methods can by default be invoked by any internal or external transaction. For interacting with another contract one either needs to create a new instance of this contract (in which case a new contract account with the functionality described in the contract class is created) or one can directly make transactions to a known contract address holding a contract of the required shape. The syntax of Solidity resembles JavaScript, enriched with additional primitives accounting for the distributed setting of Ethereum. In particular, Solidity provides primitives for accessing the transaction and the block information, like `msg.sender` for accessing the address of the account invoking the method or `msg.value` for accessing the amount of *wei* transferred by the transaction that invoked the method.

Solidity shows some particularities when it comes to transferring money to another contract especially using the provided low level functions `send` and `call`. A value transfer initiated using these functions is finally translated to an internal call transaction which implies that calling a contract might also execute code and in particular it can fail because the available gas is not sufficient for executing the code. In addition – as in the EVM – these kinds of calls do not enable exception propagation, so that the caller manually needs to checks for the return result. Another special feature of Solidity is that it allows for defining so called *fallback functions* for contracts that get executed when a call via the `send` function was performed or (using the `call` function) an address is called that however does not properly specifies the concrete function of the contract to be called.

3 Small-Step Semantics

We introduce a small-step semantics covering the full EVM bytecode, inspired by the one presented by Luu et al. [21], which we substantially revise in order to handle the missing instructions, in particular contract calls and call creation. In addition, while formalizing our semantics, we found a major flaw related to calls and several minor ones (cf. Sect. 3.7), which we fixed and reported to the authors. Due to space constraints, we refer the interested reader to the full version of the paper [22] for a formal account of the semantic rules and present below the most significant ones.

3.1 Preliminaries

In the following, we will use $\mathbb{B}$ to denote the set $\{0, 1\}$ of bits and accordingly $\mathbb{B}^x$ for sets of bitstrings of size x. We further let $\mathbb{N}_x$ denote the set of non-negative integers representable by x bits and allow for implicit conversion between those two representations. In addition, we will use the notation $[X]$ (resp. $\mathcal{L}(X)$) for arrays (resp. lists) of elements from the set X. We use standard notations for operations on arrays and lists.

3.2 Global State

As mentioned before, the global state is a (partial) mapping from account addresses (that are bitstrings of size 160) to accounts. In the case that an account does not exist, we assume it to map to $\perp$. Accounts, irrespectively of their type, are tuples of the form $(n, b, stor, code)$, with $n \in \mathbb{N}_{256}$ being the account's nonce that is incremented with every other account that the account creates, $b \in \mathbb{N}_{256}$ being the account's balance in *wei*, $stor \in \mathbb{B}^{256} \rightarrow \mathbb{B}^{256}$ being the accounts persistent storage that is represented as a mapping from 256-bit words to 256-bit words and finally $code \in [\mathbb{B}^8]$ being the contract that is an array of bytes. In contrast to contract accounts, external accounts have the empty bytearray as code. As only the execution of code in the context of the account can access and modify the account's storage, the fact that formally external accounts have

persistent storage does not have any effect. In the following, we will denote the set of addresses with $\mathcal{A}$ and the set of global states with Σ and we will assume that $\sigma \in \Sigma$.

3.3 Small-Step Relation

In order to define the small-step semantics, we give a small-step relation $\Gamma \vDash S \rightarrow S'$ that specifies how a call stack $S \in \mathbb{S}$ representing the state of the execution evolves within one step under the transaction environment $\Gamma \in \mathcal{T}_{env}$.

In Fig. 1 we give a full grammar for call stacks and transaction environments:

$$
\begin{array}{rrll}
\text{Call stacks } \mathbb{S} & \ni S & := & \mathit{EXC} :: S_{plain} \mid \mathit{HALT}(\sigma, d, g, \eta) :: S_{plain} \mid S_{plain} \\
\text{Plain call stacks } \mathbb{S}_{plain} & \ni S_{plain} & := & (\mu, \iota, \sigma, \eta) :: S_{plain} \\
\text{Machine states } M & \ni \mu & := & (\mathit{gas}, \mathit{pc}, m, i, s) \\
\text{Execution environments } I & \ni \iota & := & (\mathit{actor}, \mathit{input}, \mathit{sender}, \mathit{value}, \mathit{code}) \\
\text{Global states } \Sigma & \ni \sigma & & \\
\text{Account states } \mathbb{A} & \ni \mathit{acc} & := & (n, b, \mathit{code}, \mathit{stor}) \mid \bot \\
\text{Transaction effects } N & \ni \eta & := & (b, L, S_{\dagger}) \\
\text{Transaction environments } \mathcal{T}_{env} & \ni \Gamma & := & (o, \mathit{prize}, H)
\end{array}
$$

$$
\begin{array}{rl}
\text{Notations:} & d \in [\mathbb{B}^8], \quad g \in \mathbb{N}_{256}, \quad \eta \in N, \quad o \in \mathcal{A}, \quad \mathit{prize} \in \mathbb{N}_{256}, \quad H \in \mathcal{H} \\
& \mathit{gas} \in \mathbb{N}_{256}, \quad \mathit{pc} \in \mathbb{N}_{256}, \quad m \in \mathbb{B}^{256} \rightarrow \mathbb{B}^8 \quad i \in \mathbb{N}_{256}, \quad s \in \mathcal{L}(\mathbb{B}^{256}) \\
& \mathit{sender} \in \mathcal{A} \quad \mathit{input} \in [\mathbb{B}^8] \quad \mathit{sender} \in \mathcal{A} \quad \mathit{value} \in \mathbb{N}_{256} \quad \mathit{code} \in [\mathbb{B}^8] \\
& b \in \mathbb{N}_{256} \quad L \in \mathcal{L}(\mathit{Ev}_{\mathit{log}}) \quad S_{\dagger} \subseteq \mathcal{A} \quad \Sigma = \mathcal{A} \rightarrow \mathbb{A}
\end{array}
$$

Fig. 1. Grammar for call stacks and transaction environments

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. More specifically, the transaction environment $\Gamma \in \mathcal{T}_{env} = \mathcal{A} \times \mathbb{N}_{256} \times \mathcal{H}$ is a tuple of the form (o, prize, H) with $o \in \mathcal{A}$ being the address of the account that made the transaction, $\mathit{prize} \in \mathbb{N}_{256}$ denoting amount of wei that needs to paid for a unit of gas in this transaction and $H \in \mathcal{H}$ being the header of the block that the transaction is part of. We do not specify the format of block headers here, but just assume a set $\mathcal{H}$ of block headers.

Callstacks. A call stack S is a stack of execution states which represents the state of the execution within one internal transaction. We give a formal definition of the set of possible callstacks $\mathbb{S}$ as follows:

$$
\begin{aligned}
\mathbb{S} := \{ & \mathit{EXC} :: S_{plain}, \ \mathit{HALT}(\sigma, \mathit{gas}, d, \eta) :: S_{plain}, \ S_{plain} \\
& \mid \sigma \in \Sigma, \ \mathit{gas} \in \mathbb{N}, \ d \in [\mathbb{B}^8], \ \eta \in N, \ S_{plain} \in \mathcal{L}(M \times I \times \Sigma \times N) \}
\end{aligned}
$$

Syntactically, a call stack is a stack of regular execution states of the form $(\mu, \iota, \sigma, \eta)$ that can optionally be topped with a halting state $\mathit{HALT}(\sigma, \mathit{gas}, d, \eta)$

or an exception state *EXC*. We summarize these three types of states as execution states $\mathcal{S}$. 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. Exception states of the form *EXC* do not carry any information as in the case of an exception all effects of the terminated internal transaction are reverted and the caller state therefore stays unaffected, except for the gas. Halting states instead are of the form $HALT(\sigma, gas, d, \eta)$ specifying the global state σ the execution halted in, the gas $gas \in \mathbb{N}_{256}$ remaining from the execution, the return data $d \in [\mathbb{B}^8]$ and the additional transaction effects $\eta \in N$ of the internal transaction. The additional transaction effects carry information that are accumulated during execution, but do not influence the small-step execution itself. Formally, the additional transaction effects are a triple of the form $(b, L, S_\dagger) \in N = \mathbb{N}_{256} \times \mathcal{L}(Ev_{log}) \times \mathcal{P}(\mathcal{A})$ with $b \in \mathbb{N}_{256}$ being the refund balance that is increased by account storage operations and will finally be paid to the transaction's beneficiary, $L \in \mathcal{L}(Ev_{log})$ being the sequence of log events that the bytecode execution invoked during execution and $S_\dagger \subseteq \mathcal{A}$ being the so called suicide set – the set of account addresses that executed the SELFDESTRUCT command and therefore registered their account for deletion. The information held by the halting state is carried over to the calling state.

The state of a non-terminated internal transaction is described by a regular execution state of the form $(\mu, \iota, \sigma, \eta)$. The state is determined by the current global state σ of the system as well as the execution environment $\iota \in I$ that specifies the parameters of the current transaction (including inputs and the code to be executed), the local state $\mu \in M$ of the stack machine, and the transaction effects $\eta \in N$ collected during execution so far.

Execution Environment. The execution environment ι of an internal transaction specifies the static parameters of the transaction. It is a tuple of the form $(actor, input, sender, value, code) \in I = \mathcal{A} \times [\mathbb{B}^8] \times \mathcal{A} \times \mathbb{N}_{256} \times [\mathbb{B}^8]$ with the following components:

- $actor \in \mathcal{A}$ is the address of the account currently executing;
- $input \in [\mathbb{B}^8]$ is the data given as an input to the internal transaction;
- $sender \in \mathcal{A}$ is the address of the account that initiated the internal transaction;
- $value \in \mathbb{N}_{256}$ is the value transferred by the internal transaction;
- $code \in [\mathbb{B}^8]$ is the code currently executed.

This information is determined at the beginning of an internal transaction execution and it can be accessed, but not altered during the execution.

Machine State. The local machine state μ represents the state of the underlying state machine used for execution and is a tuple of the form (gas, pc, m, i, s) where

- $gas \in \mathbb{N}_{256}$ is the current amount of gas still available for execution;
- $pc \in \mathbb{N}_{256}$ is the current program counter;
- $m \in \mathbb{B}^{256} \rightarrow \mathbb{B}^8$ is a mapping from 256-bit words to bytes that represents the local memory;
- $i \in \mathbb{N}_{256}$ is the current number of active words in memory;
- $s \in \mathcal{L}(\mathbb{B}^{256})$ is the local 256-bit word stack of the stack machine.

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.

3.4 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.

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. Note that as the word size of the stack machine is 256, all arithmetic operations are performed modulo 2^{256}.

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

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

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 is constantly 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.

A more interesting example of a semantic rule is the one of the CALL instruction that initiates an internal call transaction. In the case of calling, several corner cases need to be treated which results in several inference rules for this case. Here, we only present one rule for illustrating the main functionality. More precisely, we present the case in that the account that should be called exists, the call stack limit of 1024 is not reached yet, and the account initiating the transaction has a sufficiently large balance for sending the specified amount of wei to the called account.

$$\frac{\begin{array}{c}\iota.code\,[\mu.\mathsf{pc}] = \mathsf{CALL} \qquad \mu.\mathsf{s} = g :: to :: va :: io :: is :: oo :: os :: s \\ \sigma(to) \neq \bot \qquad |A| + 1 < 1024 \qquad \sigma(\iota.\mathsf{actor}).\mathsf{b} \geq va \qquad aw = M\,(M\,(\mu.\mathsf{i}, io, is), oo, os) \\ c_{call} = C_{gascap}\,(va, 1, g, \mu.\mathsf{gas}) \qquad c = C_{base}\,(va, 1) + C_{mem}\,(\mu.\mathsf{i}, aw) + c_{call} \\ \mu.\mathsf{gas} \geq c \qquad \sigma' = \sigma\langle to \rightarrow \sigma(to)[\mathsf{b} \mathrel{+}= va]\rangle\langle \iota.\mathsf{actor} \rightarrow \sigma(\iota.\mathsf{actor})[\mathsf{b} \mathrel{-}= va]\rangle \\ d = \mu.\mathsf{m}\,[io, io + is - 1] \qquad \mu' = (c_{call}, 0, \lambda x.\,0, 0, \epsilon) \\ \iota' = \iota[\mathsf{sender} \rightarrow \iota.\mathsf{actor}][\mathsf{actor} \rightarrow to][\mathsf{value} \rightarrow va][\mathsf{input} \rightarrow d][\mathsf{code} \rightarrow \sigma(to).\mathsf{code}]\end{array}}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \;\rightarrow\; (\mu', \iota', \sigma', \eta) :: (\mu, \iota, \sigma, \eta) :: S}$$

For performing a call, the parameters to this call need to be specified on the machine stack. These are the amount of gas g that should be given as budget to the call, the recipient to of the call and the amount va of wei to be transferred with the call. In addition, the caller needs to specify the input data that should be given to the transaction and the place in memory where the return data of the call should be written after successful execution. To this end, the remaining arguments specify the offset and size of the memory fragment that input data should be read from (determined by io and is) and return data should be written to (determined by oo and os).

Calculating the cost in terms of gas for the execution is quite complicated in the case of CALL as it is influenced by several factors including the arguments given to the call and the current machine state. First of all, the gas that should be given to the call (here denoted by c_{call}) needs to be determined. This value is not necessarily equal to the value g specified on the stack, but also depends on the value va transferred by the call and the currently available gas. In addition, as the memory needs to be accessed for reading the input value and writing the return value, the number of active words in memory might be increased. This effect is captured by the memory extension function M. As accessing additional words in memory costs gas, this cost needs to be taken into account in the overall cost. The costs resulting from an increase in the number of active words is calculated by the function C_{mem}. Finally, there is also a base cost charged for the call that depends on the value va. As the cost also depends on the specific case for calling that is considered, the cost calculation functions receive a flag (here 1) as arguments. These technical details are spelled out in the full version [22].

The call itself then has several effects: First, it transfers the balance from the executing state (*actor* in the execution environment) to the recipient (to). To this end, the global state is updated. Here we use a special notation for the functional update on the global state using $\langle\rangle$ instead of []. Second, for initializing the execution of the initiated internal transaction, a new regular execution state

is placed on top of the execution stack. The internal transaction starts in a fresh machine state at program counter zero. This means that the initial memory is initialized to all zeros and consequently the number of active words in memory is zero as well and additionally the initial stack is empty. The gas budget given to the internal transaction is c_{call} calculated before. The transaction environment of the new call records the call parameters. This includes the sender that is the currently executing account *actor*, the new active account that is now the called account *to* as well as the value *va* sent and the input data given to the call. To this end the input data is extracted from the memory using the offset *io* and the size *is*. We use an interval notation here to denote that a part of the memory is extracted. Finally, the code in the execution environment of the new internal transaction is the code of the called account.

Note that the execution state of the caller stays completely unaffected at this stage of the execution. This is a conscious design decision in order to simplify the expression of security properties and to make the semantics more suitable to abstractions.

Besides CALL there are two different instructions for initiating internal call transactions that implement slight variations of the simple CALL instruction. These variations are called CALLCODE and DELEGATECALL, which both allow for executing another's account code in the context of the caller. The difference is that in the case of CALLCODE a new internal transaction is started and the currently executed account is registered as the sender of this transaction while in the case of DELEGATECALL an existing call is really forwarded in the sense that the sender and the value of the initiating transaction are propagated to the new internal transaction.

Analogously to the instructions for initiating internal call transactions, there is also one instruction CREATE that allows for the creation of 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 transferred value, 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. In contrast to the call transaction, a create transaction does not await a return value, but only an indication of success or failure.

For discussing how to return from an internal transaction, we show the rule for returning from a successful internal call transaction.

$$\frac{\begin{array}{c}\iota.code\,[\mu.\mathsf{pc}] = \mathsf{CALL} \qquad \mu.\mathsf{s} = g :: to :: va :: io :: is :: oo :: os :: s \\ flag = \sigma(to) = \bot\,?\,0 \,:\, 1 \qquad aw = M\,(M\,(\mu.\mathsf{i}, io, is), oo, os) \\ c_{call} = C_{gascap}\,(va, flag, g, \mu.\mathsf{gas}) \qquad c = C_{base}\,(va, flag) + C_{mem}\,(\mu.\mathsf{i}, aw) + c_{call} \\ \mu' = \mu[\mathsf{i} \rightarrow aw][\mathsf{s} \rightarrow 1 :: s][\mathsf{pc} \mathrel{+}= 1][\mathsf{gas} \mathrel{+}= gas - c][\mathsf{m} \rightarrow \mu.\mathsf{m}[[oo, oo + s - 1] \rightarrow d]]\end{array}}{\Gamma \vDash HALT(\sigma', gas, d, \eta') :: (\mu, \iota, \sigma, \eta) :: S \;\rightarrow (\mu', \iota, \sigma', \eta') :: S}$$

Leaving the caller state unchanged at the point of calling has the negative side effect that the cost calculation needs to be redone at this point in order to determine the new gas value of the caller state. But besides this, the rule is

straightforward: the program counter is incremented as usual and the number of active words in memory is adjusted as memory accesses for reading the input and return data have been made. The gas is decreased, meaning that the overall amount of gas c allocated for the execution is subtracted. However, as this cost already includes the gas budget given to the internal transaction, the gas *gas* that is left after the execution is refunded again. In addition, the return data d is written to the local memory of the caller at the place specified by *oo* and *os*. Finally, the value one is written to the caller's stack in order to indicate the success of the internal call transaction. As the execution was successful, as indicated by the halting state, the global state and the transaction effects of the callee are adopted by the caller.

EVM bytecode offers several instructions for explicitly halting (internal) transaction execution. Besides the standard instructions STOP and RETURN, there is the SELFDESTRUCT instruction that is very particular to the blockchain setting. The STOP instruction causes regular halting of the internal transaction without returning data to the caller. In contrast, the RETURN instruction allows one to specify the memory fragment containing the return data that will be handed to the caller.

Finally, the SELFDESTRUCT instruction halts the execution and lists the currently execution account for later deletion. More precisely, this means that this account will be deleted when finalizing the external transaction, but its behavior during the ongoing small-step execution is not affected. Additionally, the whole balance of the deleted account is transferred to some beneficiary specified on the machine stack.

We show the small-step rules depicting the main functionality of SELFDESTRUCT. As for CALL, capturing the whole functionality of SELFDESTRUCT would require to consider several corner cases. Here we consider the case where the beneficiary exists, the stack does not underflow and the available amount of gas is sufficient.

$$\frac{\begin{array}{c}\omega_{\mu,\iota} = \mathsf{SELFDESTRUCT} \qquad \mu.\mathsf{s} = a_{ben} :: s \\ a = a_{ben} \mod 2^{160} \qquad \sigma(a) \neq \bot \qquad \mu.\mathsf{gas} \geq 5000 \qquad g = \mu.\mathsf{gas} - 5000 \\ \sigma' = \sigma\langle \iota.\mathsf{actor} \rightarrow \sigma(\iota.\mathsf{actor})[\mathsf{balance} \rightarrow 0]\rangle\langle a \rightarrow \sigma(a)[\mathsf{balance} \mathrel{+}= \sigma.(\iota.\mathsf{actor}).\mathsf{balance}]\rangle \\ r = (\iota.\mathsf{actor} \in \Gamma.S_\dagger)\,?\,0 : 24000 \qquad \eta' = \eta[\mathsf{S}_\dagger \rightarrow \eta.\mathsf{S}_\dagger \cup \{\iota.\mathsf{actor}\}][\mathsf{balance} \mathrel{+}= r]\end{array}}{\Gamma \vDash (\mu, \iota, \sigma, \eta) :: S \rightarrow \mathit{HALT}(\sigma', g, \epsilon, \eta') :: S}$$

The SELFDESTRUCT command takes one argument a_{ben} from the stack specifying the address of the beneficiary that should get the balance of the account that is destructed. If all preconditions are satisfied, the balance of the executing account ($\iota.\mathsf{actor}$) is transferred to the beneficiary address and the current internal transaction execution enters a halting state. Additionally, the transaction effects are extended by adding $\iota.\mathsf{actor}$ to the suicide set and by possibly increasing the refund balance. The refund balance is only increased in case that $\iota.\mathsf{actor}$ is not already scheduled for deletion. The halting state captures the global state σ after the money transfer, the remaining gas g after executing the SELFDESTRUCT and the updated transaction effects η'. As no return data is handed to the caller, the empty bytearray ϵ is specified as return data in the halting state.

Note that SELFDESTRUCT deletes the currently executing account $\iota.\mathsf{actor}$ which is not necessarily the same account as the one owning the code $\iota.\mathsf{code}$. This might be due a previous execution of DELEGATECALL or CALLCODE.

3.5 Transaction Execution

The outcome of an external transaction execution does not only consist of the result of the EVM bytecode execution. Before executing the bytecode, the transaction environment and the execution environment are determined from the transaction information and the block header. In the following we assume $\mathcal{T}$ to denote the set of transactions. An (external) transaction $T \in \mathcal{T}$, similar to the internal transactions, specifies a gas limit, a recipient and a value to be transferred. In addition, it also contains the originator and the gas prize that will be recorded in the transaction environment. Finally, it specifies an input to the transaction and the transaction type that can either be a call or a create transaction. The transaction type determines whether the input will be interpreted as input data to a call transaction or as initialization code for a create transaction. In addition to the transaction of the environment initialization, some initial changes on the global state and validity checks are performed. For the sake of presentation we assume in the following a function $\mathit{initialize}(\cdot,\cdot,\cdot) \in \mathcal{T} \times \mathcal{H} \times \Sigma \to (\mathcal{T}_{env} \times \mathcal{S}) \cup \{\bot\}$ performing the initialization phase and returning a transaction environment and initial execution state in the case of a valid transaction and $\bot$ otherwise. Similarly, we assume a function $\mathit{finalize}(\cdot,\cdot,\cdot) \in T \times \mathcal{S} \times N \times \Sigma$ that given the final global state of the execution, the accumulated transaction effects and the transaction, computes the final effects on the global state. These include for example the deletion of the contracts from the suicide set and the payout to the beneficiary of the transaction.

Formally we can define the execution of a transaction $T \in \mathcal{T}$ in a block with header $H \in \mathcal{H}$ as follows:

$$\frac{\begin{array}{c}(\Gamma, s) = \mathit{initialize}(T, H, \sigma) \\ \Gamma \vDash s :: \epsilon \to^* s' :: \epsilon \qquad \mathit{final}(s') \qquad \sigma' = \mathit{finalize}(s', \eta', T)\end{array}}{\sigma \xrightarrow{T,H} \sigma'}$$

where $\to^*$ denotes the reflexive and transitive closure of the small-step relation and the predicate $\mathit{final}(\cdot)$ characterizes a state that cannot be further reduced using the small-step relation.

3.6 Formalization in F*

We provide a formalization of a large fragment of our small-step semantics in the proof assistant F* [24]. At the time of writing, we are formalizing the remaining part, which only consists of straightforward local operations, such as bitwise operators and opcodes to write code to (resp. read code from) the memory.

F* is an ML-dialect that is optimized for program verification and allows for performing manual proofs as well as automated proofs leveraging the power of SMT solvers.

Our formalization strictly follows the small-step semantics as presented in this paper. The core functionality is implemented by the function `step` that describes how an execution stack evolves within one execution state. To this end it has two possible outcomes: either it performs an execution step and returns the new callstack or – in the case that a final configuration is reached (which is a stack containing only one element that is either a halting or an exception state) – it reports the final state. In order to provide a total function for the step relation, we needed to introduce a third execution outcome that signalizes that a problem occurred due to an inconsistent state. When running the semantics from a valid initial configuration this result, however, should never be produced. For running the semantics, the function `execution` is defined that subsequently performs execution steps using `step` until reaching the final state and reports it.

The current implementation encompasses approximately thousand lines of code. Since F* code can be compiled into OCaml, we validate our semantics against the official EVM test suite [25]. Our semantics passes 304 out of 624 tests, failing only in those involving any of the missing functionalities.

We make the formalization in F* publicly available [22] in order to facilitate the design of static analysis techniques for EVM bytecode as well as their soundness proofs.

3.7 Comparison with the Semantics by Luu et al. [21]

The small-step semantics defined by Luu et al. [21] encompasses only a variation of a subset of EVM bytecode instructions (called EtherLite) and assumes a heavily simplified execution configuration. The instructions covered span simple stack operations for pushing and popping values, conditional branches, binary operations, instructions for accessing and altering local memory and account storage, as well as as the ones for calling, returning and destructing the account. Essential instructions as CREATE and those for accessing the transaction and block information are omitted. The authors represent a configuration as a tuple of a call stack of activation records and the global state. An activation record contains the code to be executed, the program counter, the local memory and the machine stack. The global state is modelled as mapping from addresses to accounts, with the latter consisting of code, balance and persistent storage.

The overall abstraction contains a conceptual flaw, as not including the global state in the activation records of the call stack does not allow for modelling that, in the case of an exception in the execution of the callee, the global state is rolled back to the one of the caller at the point of calling. In addition, the model cannot be easily extended with further instructions – such as further call instructions or instructions accessing the environment – without major changes in the abstraction as a lot of information, e.g., the one captured in our small-step semantics in the transaction and the execution environment, are missing.

4 Security Definitions

In the following, we introduce the semantic characterization of the most significant security properties for smart contracts, motivating them with typical vulnerabilities recurring in the wild.

For selecting those properties, we inspected the classification of bugs performed in [13,21]. To our knowledge, these are the only works published so far that aim at systematically summarizing bugs in Ethereum smart contracts.

For the presented bugs, we synthesized the semantic security properties that were violated. In this process we realized that some bugs share the same underlying property violation and that other bugs can not be captured by such generic properties – either because they are of a purely syntactic nature or because they constitute a derivation from a desired behavior that is particular to a specific contract.

Preliminary Notations. Formally, we represent a contract as a tuple of the form $(a, code)$ where $a \in \mathcal{A}$ denotes the address of the contract and $code \in [\mathbb{B}]$ denotes the contract's code. We denote the set of contracts by $\mathcal{C}$ and assume functions $address(\cdot)$ and $code(\cdot)$ that extract the contract address and code respectively.

As we will argue about contracts being called in an arbitrary setting, we additionally introduce the notion of *reachable configuration*. Intuitively, a pair (Γ, S) of a transaction environment Γ and a call stack S is reachable if there exists a state s such that S, s are the result of $initialize(T,\ H,\ \sigma)$, for some transaction T, block header H, a global state σ, and S is reachable from s.

Definition 1 (Reachable Configuration). *The pair $(\Gamma, A) \in \mathcal{T}_{env} \times \mathcal{S}$ is a reachable configuration if for some transaction $T \in \mathcal{T}$, some block header $H \in \mathcal{H}$ and some global state $\sigma \in \mathcal{A} \to \mathbb{A}$ of the blockchain it holds that*

$$(\Gamma, s) = initialize(T, H, \sigma) \wedge \Gamma \vDash s :: \epsilon \ \rightarrow^* S$$

In order to give concise security definitions, we further introduce, and assume throughout 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 $\perp$. Specifically, we let

$$\begin{aligned}\mathbb{S}_n := \{&EXC_c :: S_{plain},\ HALT(\sigma, gas, \eta, d)_c :: S_{plain},\ S_{plain} \\ &\mid \sigma \in \Sigma,\ gas \in \mathbb{N},\ d \in [\mathbb{B}^8],\ \eta \in N,\ S_{plain} \in \mathcal{L}((M \times I \times \Sigma \times N) \times \mathcal{C})\}\end{aligned}$$

where $c \in \mathcal{C} \cup \{\perp\} = \mathcal{C}_\perp$.

Next, we introduce the notion of execution trace for smart contract execution. Intuitively, a trace is a sequence of actions. In our setting, the actions to be recorded are composed of an opcode, the address of the executing contract, and a sequence of arguments to the opcode. We denote the set of actions with *Act*. Accordingly, every small step produces a trace consisting of a single action. Again, we lift the resulting trace semantics to multiple execution steps that then

produce sequences of actions $\pi \in \mathcal{L}(Act)$. We only report the trace semantics definition for the CALL case here, referring to the full version of the paper for the details [22].

$$\frac{\iota.code[\mu.\mathsf{pc}] = \mathsf{CALL} \qquad \mu.\mathsf{s} = g :: to :: va :: io :: is :: oo :: os :: s \quad \cdots \quad \mu' = \cdots \quad \iota' = \cdots \quad \sigma' = \cdots}{\Gamma \vDash (\mu, \iota, \sigma)_c :: S \xrightarrow{\mathsf{CALL}_c(g,to,io,is,oo,os)} (\mu', \iota', \sigma')_{to} :: (\mu, \iota, \sigma)_c :: S}$$

We will write $\pi \downarrow_{\mathsf{calls}_c}$ to denote the projection of π to calls performed by contract c, i.e., actions of the form $\mathsf{CALL}_c(g, to, va, io, is, oo, os)$, $\mathsf{CREATE}_c(va, io, is)$, $\mathsf{CALLCODE}_c(g, to, va, io, is, oo, os)$, and $\mathsf{DELEGATECALL}_c(g, to, io, is, oo, os)$.

4.1 Call Integrity

Dependency on Attacker Code. One of the most famous bugs of Ethereum's history is the so called DAO bug that led to a loss of 60 million dollars in June 2016 [10]. This bug is in the literature classified as reentrancy bug [13,21] as the affected contract was drained out of money by subsequently reentering it and performing transactions to the attacker on behalf of the contract. More generally, the problem of this contract was that malicious code was able to affect the outgoing money flows 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, the `call` construct invokes the called contract's fallback function when no particular function of the contract is specified (2). Consequently, 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 following contracts:

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

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

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.

Call Integrity. In order to protect from this class of bugs, it is crucial to secure the code against being reentered before regaining control over the control flow. From a security perspective, the fundamental problem is that the contract behaviour depends on untrusted code, even though this was not intended by the developer. We capture this intuition through a hyperproperty, which we name *call integrity*. The idea is that no matter how the attacker can schedule c (callstacks S and S' in the definition), the calls of c (traces π, π') cannot be controlled by the attacker, even if c hands over the control to the attacker.

Definition 2 (Call Integrity). *A contract $c \in \mathcal{C}$ satisfies call integrity for a set of addresses $\mathcal{A}_C \subseteq \mathcal{A}$ if for all reachable configurations $(\Gamma, s_c :: S), (\Gamma, {s'}_c :: S')$ with s, s' differing only in the code with address in $\mathcal{A}_C$, it holds that for all t, t'*

$$\Gamma \vDash s_c :: S \xrightarrow{\pi}{}^{*} t_c :: S \ \wedge\ \mathit{final}(t_c) \ \wedge\ \Gamma \vDash {s'}_c :: S' \xrightarrow{\pi'}{}^{*} {t'}_c :: S' \ \wedge\ \mathit{final}({t'}_c) \implies \pi \downarrow_{\mathit{calls}_c} = \pi' \downarrow_{\mathit{calls}_c}$$

4.2 Proof Technique for Call Integrity

We now establish a proof technique for call integrity, based on local properties that are arguably easier to verify and that we show to imply call integrity. As a first observation, we identify the different ways in which external contracts can influence the execution of a smart contract c and introduce corresponding security properties:

Code Dependency. The contract c might access (information on) the untrusted contracts code via the EXTCODECOPY or the EXTCODESIZE instructions and make his behaviour depend on those values;

Effect Dependency. The contract c might call the untrusted contract and might depend on its execution effects and return value;

Re-entrancy. The contract c might call the untrusted contract, with the latter influencing the behaviour of the former by performing changes to the global state itself or "on behalf" of c by reentering it and thereby potentially decreasing the balance of c.

The first two of these properties can be seen as value dependencies and therefore can be formalized as hyperproperties. The first property says that the calls performed by a contract should not be affected by the effects on the execution state produced by adversarial contracts. Technically, we consider a contract c calling an adversarial contract c' (captured as $\Gamma \vDash s_c :: S \rightarrow {s''}_{c'} :: s_c :: S$ in the premise), which we let terminate in two arbitrary states s', t': we require that c's continuation code performs the same calls in both states.

Definition 3 ($\mathcal{A}_C$-effect Independence). *A contract $c \in \mathcal{C}$ is $\mathcal{A}_C$-effect independent of for a set of addresses $\mathcal{A}_C \subseteq \mathcal{A}$ if for all reachable configurations $(\Gamma, s_c :: S)$ such that $\Gamma \vDash s_c :: S \rightarrow s''_{c'} :: s_c :: S$ for some s'' and address$(c') \in \mathcal{A}_C$, it holds that for all final states s', t' whose global state might differ in all components but the code from the global state of s,*

$$\begin{aligned} \Gamma_{init} \vDash s'_{c'} :: s_c :: S \xrightarrow{\pi}{}^{*} s''_c :: S \ \wedge \ final(s'') \\ \wedge \ \Gamma_{init} \vDash t'_{c'} :: s_c :: S \xrightarrow{\pi'}{}^{*} t''_c :: S \wedge final(t'') \\ \implies \pi \downarrow_{calls_c} = \pi' \downarrow_{calls_c} \end{aligned}$$

The second property says that the calls of a contract should not be affected by the code read from the blockchain (e.g., the code does not branch on code read from the blockchain). To this end we introduce the notation $\Gamma \vdash s :: S \xrightarrow[f]{\pi}{}^{*} s' :: S$ to denote that the local small-step execution of state s on stack S under Γ results in several steps in state s' producing trace π given that in the local execution steps of EXTCODECOPY and EXTCODESIZE, which are the operations used to access the code on the global state, the code returned by these functions is determined by the partial function $f \in \mathcal{A} \rightharpoonup [\mathbb{B}]$ as opposed to the global state. In other words, we consider in the premise a contract c reading two different codes from the blockchain and terminating in both runs (captured as $\Gamma \vdash s_c :: S \xrightarrow[f]{\pi}{}^{*} s'_c :: S$ and $\Gamma \vdash s_c :: S \xrightarrow[f']{\pi'}{}^{*} s''_c :: S$), and we require that c performs the same calls in both runs.

Definition 4 ($\mathcal{A}_C$-code Independence). *A contract $c \in \mathcal{C}$ is $\mathcal{A}_C$-code independent for a set of addresses $\mathcal{A}_C \subseteq \mathcal{A}$ if for all reachable configurations $(\Gamma, s_c :: S)$ it holds for all local code updates $f, f' \in \mathcal{A} \rightharpoonup [\mathbb{B}]$ on $\mathcal{A}_C$ that*

$$\begin{aligned} \Gamma \vdash s_c :: S \xrightarrow[f]{\pi}{}^{*} s'_c :: S \ \wedge \ final(s') \ \wedge \ \Gamma \vdash s_c :: S \xrightarrow[f']{\pi'}{}^{*} s''_c :: S \ \wedge \ final(s'') \\ \implies \pi \downarrow_{calls_c} = \pi' \downarrow_{calls_c} \end{aligned}$$

Both these independence properties can be overapproximated by static analysis techniques based on program dependence graphs [26], as done by Joana to verify non-interference in Java [27]. The idea is to traverse the dependence graph in order to detect dependencies between the sensitive sources, in our case the data controlled by the adversary and returned to the contract, and the observable sinks, in our case the local contract calls.

The last property constitutes a safety property. Specifically, single-entrancy states that it cannot happen that when reentering the contract c another call is performed before returning (i.e., after reentrancy, which we capture in the call stack as two distinct states with the same running contract c, the call stack cannot further increase).

Definition 5 (Single-entrancy). *A contract $c \in \mathcal{C}$ is single-entrant if for all reachable configurations $(\Gamma, s_c :: S)$, it holds for all s', s'', S' that*

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

This safety property can be easily overapproximated by syntactic conditions, as for instance done in the Oyente analyzer [21].

Finally, the next theorem proves the soundness of our proof technique, i.e., the two independence properties and the single-entrancy property together entail call integrity.

Theorem 1. *Let $c \in \mathcal{C}$ be a contract and $\mathcal{A}_C \subseteq \mathcal{A}$ be a set of untrusted addresses. If c is $\mathcal{A}_C$-local independent, c is $\mathcal{A}_C$-effect independent, and c is single-entrant then c provides call integrity for $\mathcal{A}_C$.*

Proof Sketch. Let $(\Gamma, s_c :: S), (\Gamma, {s'}_c :: S')$ be reachable configurations such that s, s' differ only in the code with address in $\mathcal{A}_C$. We now compare the two small-step runs of those configurations. Due to $\mathcal{A}_C$-code independence, the execution until the first call to an address $a \in \mathcal{A}_C$ produces the same partial trace until the call to a. Indeed, we can express the runs under different address mappings through the code update from the $\mathcal{A}_C$-code independence property, as long as no call to one of the updated addresses is performed. When a first call to $a \in \mathcal{A}_C$ is performed, we know due to single-entrancy that the following call cannot produce any partial execution trace for any of the runs as this would imply that contract c is reentered and a call out of the contract is performed. Due to $\mathcal{A}_C$-code independence and $\mathcal{A}_C$-effect independence , the traces after returning must coincide till the next call to an address in $\mathcal{A}_C$. This argument can be iteratively applied until reaching the final state of the execution of c.

4.3 Atomicity

Exception Handling. As discussed in Sect. 2, the way exceptions are propagated varies with the way contracts are called. In particular, in the case of `call` and `send`, exceptions are not propagated, but a manual check for the successful completion of the called function's execution is required. This behavior reflects the way exceptions are reported during bytecode execution: Instead of propagating up through the call stack, the callee reports the exception to the caller by writing zero to the stack. In the context of Ethereum, the issue of exception handling is particularly delicate as due to the gas restriction, it might always happen that a call fails simply because it ran out of gas. Intuitively, a user would expect a contract not to depend on the concrete gas value that is given to it, with the exception that a contract might always fail completely (and consequently does not perform any changes on the global state). Such a behavior would prevent contracts from entering an inconsistent state as the one presented in the following excerpt of a simple banking contract:

```
1  contract SimpleBank{mapping( address => uint) balances;
2    function withdraw(){ msg.sender.send(balances[msg.sender]));
3      balances[msg.sender] = 0;}}
```

The contract keeps a record of the user balances and provides a function that allows a user to withdraw its own balance – which results in an update of the record. A developer might not expect that the `send` might fail, but as it is on the bytecode level represented by a CALL instruction, additional to the Ether transfer, code might be executed that runs out of gas. As a consequence, the contract would end up in a state where the money was not transferred (as all effects of the call are reverted in case of an exception), but still the internal balance record of the contract was updated and consequently the money cannot be withdrawn by the owner anymore.

Inspired by such situations where an inconsistent state is entered by a contract due to mishandled gas exceptions, we introduce the notion of *atomicity* of a contract. Intuitively, atomicity requires that the effects of the execution on the global state do not depend on the amount of gas available – except when an exception is triggered, in which case the overall execution should have no effect at all. The last condition is captured by requiring that the final global state is the same as the initial one for at least one of the two executions (intuitively, the one causing the exception).

Definition 6. *A contract $c \in \mathcal{C}$ satisfies atomicity if for all reachable configurations (Γ, S') such that $\Gamma \vDash S' \rightarrow s_c :: S$, it holds for all gas values $g, g' \in \mathbb{N}_{256}$ that*

$$\begin{aligned} \Gamma \vDash s_c[\mu.\mathit{gas} \rightarrow g] :: S \rightarrow^* {s'}_c :: S \ \wedge \ \mathit{final}(s') \\ \wedge \ \Gamma \vDash s_c[\mu.\mathit{gas} \rightarrow g'] :: S \rightarrow^* {s''}_c :: S \ \wedge \ \mathit{final}(s'') \\ \implies s'.\sigma = s''.\sigma \vee s.\sigma = s'.\sigma \vee s.\sigma = s''.\sigma \end{aligned}$$

4.4 Independence of Miner Controlled Parameters

Another particularity of the distributed blockchain environment is that users while performing transactions cannot make assumptions on large parts of the context their transaction will be executed in. A part of this is due to the asynchronous nature of the system: it can always be that another transaction that alters the context was performed first. Actually, the situation is even more delicate as transactions are not processed in a first-come-first-serve manner, but miners have a big influence on the execution context of transactions. They can decide upon the order of the transactions in a block (and also sneak their own transactions in first) and in addition they can even control some parameters as the block timestamp within a certain range. Consequently, contracts whose (outgoing) money flows depend either on miner controlled block information or on state information (as the state of their storage or their balance) that might be changed by other transactions are prone to manipulations by miners. A typical example adduced in the literature is the use of block timestamps as source of randomness [13,21]. In a classical lottery implementation that randomly pays

out to one of the participants and uses the block timestamp as source of randomness, a malicious miner can easily influence the result in his favor by selecting a beneficial timestamp.

We capture the absence of the miner's influence by two definitions, one saying that the outgoing Ether flows of a contract should not be influenced by components of the transaction environment that can be (within a certain range) set by miners and the other one saying that the Ether flows should not depend on those parts of the contract state that might have been influenced by previously executed transactions. The first definition rules out what is in the literature often described as timestamp dependency [13,21].

First, we define *independence of* (parts of) *the transaction environment*. To this end, we assume $\mathcal{C}_\Gamma$ to be the set of components of the transaction environment and write $\Gamma =_{/c_\Gamma} \Gamma'$ to denote that the transaction environments Γ, Γ' are equal up to component c_Γ.

Definition 7 (Independence of the Transaction Environment). *A contract $c \in \mathcal{C}$ is independent of a subset $I \subseteq \mathcal{C}_\Gamma$ of components of the transaction environment if for all $c_\Gamma \in I$ and all reachable configurations $(\Gamma, s_c :: S)$ it holds for all Γ' that*

$$c_\Gamma(\Gamma) \neq c_\Gamma(\Gamma') \wedge \Gamma =_{/c_\Gamma} \Gamma'$$
$$\wedge\, \Gamma \vDash s_c :: S \xrightarrow{\pi}{}^{*} {s'}_c :: S \;\wedge\; \mathit{final}(s') \;\wedge\; \Gamma' \vDash s_c :: S \xrightarrow{\pi'}{}^{*} {s''}_c :: S \;\wedge\; \mathit{final}(s'')$$
$$\implies \pi \downarrow_{\mathsf{calls}_c} = \pi' \downarrow_{\mathsf{calls}_c}$$

Next, we define the notion of *independence of the account state*. Formally, we capture this property by requiring that the outgoing Ether flows of the contract under consideration should not be affected by those parameters of the contract that might have been changed by previous executions which are the balance, the account's nonce, and the account's persistent storage.

Definition 8 (Independence of Mutable Account State). *A contract $c \in \mathcal{C}$ is independent of the account state if for all reachable configurations $(\Gamma, s_c :: S), (\Gamma, s_c :: S')$ with s, s' differing only in the nonce, balance and storage for address (c), it holds that*

$$\Gamma \vDash s_c :: S \xrightarrow{\pi}{}^{*} {s'}_c :: S \;\wedge\; \mathit{final}({s'}_c) \;\wedge\; \Gamma \vDash s_c :: S' \xrightarrow{\pi'}{}^{*} {s''}_c :: S \;\wedge\; \mathit{final}({s''}_c)$$
$$\implies \pi \downarrow_{\mathsf{calls}_c} = \pi' \downarrow_{\mathsf{calls}_c}$$

As far the other independence properties, both these properties can be statically verified using program dependence graphs.

4.5 Classification of Bugs

The previously presented security definitions are motivated by the bugs that were observed in real Ethereum smart contracts and studied in [13,21]. Table 1 gives an overview on the bugs from the literature that are ruled out by our security properties.

Table 1. Bugs from [13,21] ruled out by the security properties

Security property	Bug
Call integrity	Reentrancy [13,21]
	Call to the unknown [13]
Atomicity	Mishandled exceptions [13,21]
Independence of mutable account state	Transaction order dependency [21]
	Unpredictable state [13]
Independence of transaction environment	Timestamp dependancy [21]
	Time constraints [13]
	Generating randomness [13]

Our security properties do not cover all bugs described by Atzei et al. [13], as some of the bugs do not constitute violations of general security properties, i.e., properties that are not specific to the particular contract implementation. There are two classes of bugs that we do not consider: The first class deals with the occurrence of unexpected exceptions (such as the Gasless Send and the Call stack Limit bug) and the second class encompasses bugs caused by the Solidity semantics deviating from the programmer's intuitions (such as the Keeping Secrets, Type Cast and Exception Disorders bugs).

The first class of bugs encompasses runtime exceptions that are hard to predict for the developer and that are consequently not handled correctly. Of course, it would be possible to formalize the absence of those particular kinds of exceptions as simple reachability properties using the small-step semantics. Still, such properties would not give any insight about the security of a contract: the fact that a particular exception occurs can be unproblematic in the case that proper exception handling is in place. In general, the notion of a correct exception handling highly depends on the specific contract's intended behavior. For the special case of out-of-gas exceptions, we could introduce the notion of atomicity in order to capture a generic goal of proper exception handling. But such a notion is not necessarily sufficient for characterizing reasonable ways of dealing with other kinds of runtime exceptions.

The second class of bugs are introduced on the Solidity level and are similarly hard to account for by using generic security properties. Even though these bugs might all originate from similar idiosyncrasies of the Solidity semantics, the impact of the bugs on the contract's semantics might deviate a lot. This

might result in violations of the security properties discussed before, but also in violating the contract's functional correctness. Consequently, catching those bugs might require the introduction of contract-specific correctness properties.

Finally, Atzei et al. [13] discuss the Ether Lost in Transfer bug. This bug is introduced by sending Ether to addresses that do not belong to any contract or user, so called orphan addresses. We could easily formalize a reachability property stating that no valid contract execution should ever send Ether to such an address. We omit such a definition here as it is quite straightforward and at the same time it is not a property that directly affects the security of an individual contract: Sending Ether to such an orphan address might have negative impacts on the overall system as money is effectively lost. For the specific contract sending this money, this bug can be seen as a corner case of sending Ether to an unintended address which rather constitutes a correctness violation.

4.6 Discussion

As previously discussed, we are not aware of any prior formal security definitions of smart contracts. Nevertheless, we compared our definitions with the verification conditions used in Oyente [21]. Our investigation shows that the verification conditions adopted in this tool are neither sound nor complete.

For detecting mishandled exceptions, it is checked whether each CALL instruction in the contract code is directly followed by the ISZERO instruction that checks whether the top element of the stack is zero. Unfortunately, Oyente (although stated in the paper) does not implement this check, so that we needed to manually inspect the bytecodes for determining the outcomes of the syntactic check. As shown in Fig. 2a a check for the caller returning zero does not necessarily imply a proper exception handling and therefore atomicity of the contract. This excerpt of a simple banking contract that keeps track of the users' balances and allows users to withdraw their balances using the function `withdraw` checks for the success of the performed call, but still does not react accordingly. It only makes sure that the number of successes is updated consistently, but does not perform the update on the user's balance record according to the call outcome.

On the other hand, not performing the desired check does not imply the absence of atomicity as illustrated in Fig. 2b. Writing the outcome in some variable before checking it, satisfies the negative pattern, but still correct exception handling is performed. For detecting timestamp dependency, Oyente checks whether the contract has a symbolic execution path with the timestamp (that is represented as own symbolic variable) being included in one of its constraints. This definition however, does not capture the case shown in Fig. 2c.

This contract is clearly timestamp dependent as whether or not the function `pay` pays out some money to the sender depends on the timestamp set when creating the contract. A malicious miner could consequently manipulate the block timestamp for a transaction that creates such a contract in a way that money is paid out and then subsequently query it for draining it out. This is however, not captured by the characterization of the property in Oyente as they only capture the local execution paths of the contract.

```
contract SimpleBank{
 mapping( address => uint) bal;
 uint successes;
  function withdraw(){
   if (msg.sender.send(bal[msg.sender]))
    {  successes++; }
   bal[msg.sender] = 0;}}
```

(a)

```
contract SimpleBank{
 mapping( address => uint) bal;
  function withdraw(){
    bool b =
    msg.sender.send(bal[msg.sender]);
    if (b) bal[msg.sender] = 0;}}
```

(b)

```
contract Test{
 uint time = block.timestamp;
  function pay (){
   if (time % 2 == 1){
    msg.sender.send(100);}}}
```

(c)

```
contract Test {
 function pay (){
  if (block.timestamp % 2 == 1 ||
   block.timestamp % 2 == 0){
    msg.sender.send(100);}}}
```

(d)

```
contract Fund{
 mapping( address => uint) shares;
  function withdraw(){
   if (msg.sender.send(shares[msg.sender]))
    shares[msg.sender] = 0;}}
```

(e)

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

(f)

Fig. 2. (a) Exception handling: false negative (b) Exception handling: false positive (c) Timestamp dependency: false negative (d) Timestamp dependency: false positive (e) Reentrancy: false negative (f) Reentrancy: false positive

On the other hand, using the block timestamp in path constraints does not imply a dependency as can easily be seen by the example in Fig. 2d.

For the transaction order dependency and the reentrancy property, we were unfortunately not able to reconcile the property characterization provided in the paper with the implementation of Oyente.

For checking reentrancy according to the paper, it should be checked whether the constraints on the path leading to a CALL instruction can still be satisfied after performing the updates on the path (e.g. changing the storage). If so, the contract is flagged as reentrant. According to our understanding, this approach should not flag contracts that correctly guard their calls as reentrant. Still, by the version of Oyente provided with the paper the contract in Fig. 2f is tagged as reentrant.

There exists an updated version of Oyente [28] that is able to precisely tag this contract as not reentrant, but we could not find any concrete information on the criteria used for checking this property. Still, we found out that the underlying characterization can not be sufficient for detecting reentrancy as the contract in Fig. 2e is classified not to exhibit a reentrancy vulnerability even though it should as the `send` command also executes the recipient's callback function (even though with limited gas). The example is taken from the Solidity documentation [23] where it is listed as negative example. For transaction order dependency, Oyente

should check whether execution traces exhibiting different Ether flows exists. But it turned out that not even a simple example of a transaction dependent contract can be detected by any of the versions of Oyente.

5 Conclusions

We presented the first complete small-step semantics of EVM bytecode and formalized a large fragment thereof in the F* proof assistant, successfully validating it against the official Ethereum test suite. We further defined for the first time a number of salient security properties for smart contracts, relying on a combination of hyper- and safety properties. Our framework is available to the academic community in order to facilitate future research on rigorous security analysis of smart contracts.

In particular, this work opens up a number of interesting research directions. First, it would be interesting to formalize in F* the semantics of Solidity code and a compiler from Solidity into EVM, formally proving its soundness against our semantics. This would allow us to provide software developers with a tool to verify the security of their code, from which they could obtain bytecode that is secure by construction. Second, we intend to design an efficient static analysis technique for EVM bytecode and to formally prove its soundness against our semantics.

Acknowledgments. This work has been partially supported by the European Research Council (ERC) under the European Union's Horizon 2020 research (grant agreement No 771527-BROWSEC), by Netidee through the project EtherTrust (grant agreement 2158), and by the Austrian Research Promotion Agency through the Bridge-1 project PR4DLT (grant agreement 13808694).

References

1. Nakamoto, S.: Bitcoin: a peer-to-peer electronic cash system (2008). http://bitcoin.org/bitcoin.pdf
2. Biryukov, A., Khovratovich, D., Tikhomirov, S.: Findel: secure derivative contracts for Ethereum. In: Brenner, M., Rohloff, K., Bonneau, J., Miller, A., Ryan, P.Y.A., Teague, V., Bracciali, A., Sala, M., Pintore, F., Jakobsson, M. (eds.) FC 2017. LNCS, vol. 10323, pp. 453–467. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70278-0_28. http://orbilu.uni.lu/bitstream/10993/30975/1/Findel_2017-03-08-CR.pdf
3. Hahn, A., Singh, R., Liu, C.C., Chen, S.: Smart contract-based campus demonstration of decentralized transactive energy auctions. In: 2017 IEEE Power and Energy Society Innovative Smart Grid Technologies Conference (ISGT), pp. 1–5. IEEE (2017)
4. McCorry, P., Shahandashti, S.F., Hao, F.: A smart contract for boardroom voting with maximum voter privacy. In: Kiayias, A. (ed.) FC 2017. LNCS, vol. 10322, pp. 357–375. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70972-7_20
5. Adhikari, C.: Secure framework for healthcare data management using Ethereum-based blockchain technology (2017)

6. Notheisen, B., Gödde, M., Weinhardt, C.: Trading stocks on blocks-engineering decentralized markets. In: Maedche, A., vom Brocke, J., Hevner, A. (eds.) DESRIST 2017. LNCS, vol. 10243, pp. 474–478. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-59144-5_34
7. Mathieu, F., Mathee, R.: Blocktix: decentralized event hosting and ticket distribution network (2017). https://blocktix.io/public/doc/blocktix-wp-draft.pdf
8. Azaria, A., Ekblaw, A., Vieira, T., Lippman, A.: MedRec: using blockchain for medical data access and permission management. In: International Conference on Open and Big Data (OBD), pp. 25–30. IEEE (2016)
9. Dong, C., Wang, Y., Aldweesh, A., McCorry, P., van Moorsel, A.: Betrayal, distrust, and rationality: smart counter-collusion contracts for verifiable cloud computing (2017)
10. The DAO smart contract (2016). http://etherscan.io/address/0xbb9bc244d798123fde783fcc1c72d3bb8c189413#code
11. The parity wallet breach (2017). https://www.coindesk.com/30-million-ether-reported-stolen-parity-wallet-breach/
12. The parity wallet vulnerability (2017). https://paritytech.io/blog/security-alert.html
13. Atzei, N., Bartoletti, M., Cimoli, T.: A survey of attacks on Ethereum smart contracts (SoK). In: Maffei, M., Ryan, M. (eds.) POST 2017. LNCS, vol. 10204, pp. 164–186. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54455-6_8
14. Wood, G.: Ethereum: a secure decentralised generalised transaction ledger. Ethereum Project Yellow Paper **151** (2014). https://ethereum.github.io/yellowpaper/paper.pdf
15. Hirai, Y.: Defining the Ethereum virtual machine for interactive theorem provers. In: Brenner, M., Rohloff, K., Bonneau, J., Miller, A., Ryan, P.Y.A., Teague, V., Bracciali, A., Sala, M., Pintore, F., Jakobsson, M. (eds.) FC 2017. LNCS, vol. 10323, pp. 520–535. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70278-0_33
16. Hildenbrandt, E., Saxena, M., Zhu, X., Rodrigues, N., Daian, P., Guth, D., Rosu, G.: KEVM: a complete semantics of the Ethereum virtual machine. http://hdl.handle.net/2142/97207
17. Stefănescu, A., Park, D., Yuwen, S., Li, Y., Roşu, G.: Semantics-based program verifiers for all languages. In: Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 74–91. ACM (2016)
18. Sergey, I., Hobor, A.: A concurrent perspective on smart contracts. arXiv preprint arXiv:1702.05511 (2017)
19. Mavridou, A., Laszka, A.: Designing secure ethereum smart contracts: a finite state machine based approach. http://aronlaszka.com/papers/mavridou2018designing.pdf
20. Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Gollamudi, A., Gonthier, G., Kobeissi, N., Kulatova, N., Rastogi, A., Sibut-Pinote, T., Swamy, N., Zanella-Béguelin, S.: Formal verification of smart contracts: short paper. In: Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security, pp. 91–96. ACM (2016)
21. 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, pp. 254–269. ACM (2016)

22. Grishchenko, I., Maffei, M., Schneidewind, C.: A semantic framework for the security analysis of Ethereum smart contracts. Technical report (2018). https://secpriv.tuwien.ac.at/tools/ethsemantics
23. Solidity documentation. http://solidity.readthedocs.io/en/develop/
24. F*. https://fstar-lang.org
25. Consensus test suite. https://github.com/ethereum/tests
26. Hammer, C., Snelting, G.: Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. Int. J. Inf. Secur. **8**(6), 399–422 (2009)
27. Snelting, G., Giffhorn, D., Graf, J., Hammer, C., Hecker, M., Mohr, M., Wasserrab, D.: Checking probabilistic noninterference using JOANA. IT - Inf. Technol. **56**, 280–287 (2014)
28. Luu, L., Chu, D.H., Olickel, H., Saxena, P., Hobor, A.: An analysis tool for smart contracts. https://github.com/melonproject/oyente