Solidifier: bounded model checking Solidity using lazy contract deployment and precise memory modelling

Pedro Antonino
pedro@tbtl.com
The Blockhouse Technology Limited
Oxford, UK

A. W. Roscoe
awroscoe@gmail.com
The Blockhouse Technology Limited
University College Oxford Blockchain Research Centre
Department of Computer Science, Oxford University
Oxford, UK

ABSTRACT

The exploitation of smart-contract vulnerabilities can lead to catastrophic losses. Formal verification can be a useful tool in identifying these vulnerabilities before deployment. We present an encoding of Solidity and the Ethereum blockchain using Boogie, an intermediate verification language. Based on this formalisation, we create Solidifier: a bounded model checker for Solidity. Distinctive features of our encoding are precisely capturing Solidity's unorthodox memory model, a notion of lazy blockchain exploration, and memory-precise verification harnesses. Unlike much of the work in this area, our *modus operandi* is not matching contracts against specific known behavioural patterns that might lead to vulnerabilities. Rather, we provide a tool to find errors/bad states - be they vulnerabilities or not - that might be reached through behaviours that might not follow such a pattern.

CCS CONCEPTS

• **Software and its engineering** → **Formal software verification**; **Formal language definitions**; **Software functional properties**.

KEYWORDS

Blockchain, Smart contracts, Solidity, Ethereum, Formal Verification, Bounded model checking, Boogie, Corral

ACM Reference Format:
Pedro Antonino and A. W. Roscoe. 2021. Solidifier: bounded model checking Solidity using lazy contract deployment and precise memory modelling. In *The 36th ACM/SIGAPP Symposium on Applied Computing (SAC '21), March 22–26, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3412841.3442051

SAC '21, March 22–26, 2021, Virtual Event, Republic of Korea

ACM ISBN 978-1-4503-8104-8/21/03…$15.00
https://doi.org/10.1145/3412841.3442051

1 INTRODUCTION

Smart contracts provide a new paradigm for *trusted* execution of code [2, 42]. A *smart contract* is a program that runs in a trusted abstract machine/execution environment and has access to trusted data; its behaviour typically involves managing some digital asset. To achieve this, such a program is typically run in the context of a blockchain [2, 3, 11], but other technologies can be used [17, 27, 46]. A blockchain is a decentralised system where all nodes maintain a consistent view over its database and transactions are processed in order to alter the state of this database. Smart contracts can also be seen as a way to extend the capabilities of a blockchain: the code of a smart contract can be used to encode some transaction validation logic that was not originally enforced by the blockchain [3, 11]. Being a program, it can have vulnerabilities that expose the digital assets it manages. When exploited, these flaws can lead to catastrophic effects such as the loss of vast sums of money [13]. The ability to reliably write transaction logic using smart contracts, and the flexibility that comes with it, will play a key role in the dissemination and adoption of blockchains and in the realisation of all their expected/predicted impact in society [19, 26, 41].

Formal methods have been used in many contexts to ensure that systems behave as expected [12, 14, 15, 18]. Then, it is only natural for them to be employed for the specification and verification of smart contracts [8, 9, 16, 21–24, 28, 31, 32, 44, 45]. Most of these approaches focus on the Ethereum Virtual Machine (EVM) bytecode [2], the low-level language used by the Ethereum blockchain [2], and on finding specific patterns of code that might lead to a vulnerability using symbolic execution or static analysis. In this paper, we propose a framework to analyse (in our case falsify) semantic properties of programs in Solidity [6], a high-level language used by Ethereum smart-contract developers. Like [23, 37, 45], we advocate the analysis of user-specified semantic properties of contracts, since their violations can exhibit known but also *unanticipated* behavioural patterns leading to a vulnerability. Furthermore, this can be used to analyse the functional correctness of smart contracts.

We address the following challenges in formalising and verifying Solidity smart contracts executing on the Ethereum blockchain: (1) capturing Solidity's unorthodox memory model, (2) dealing with inter-contract calls, (3) devising precise verification harnesses. In Solidity, the memory space available to a contract is divided into a persistent part called the *storage*

```
1  contract Wallet {
2    enum Status { None, Open, Closed }
3    struct Account { uint coins; Status status; }
4    mapping (address => Account) accs;
5
6    ... open, close, deposit functions omitted ...
7
8    function withdraw (uint value) public {
9      require(accs[msg.sender].status == Status.Open
           );
10     require(accs[msg.sender].coins >= value);
11     Account memory acc_mem;
12     acc_mem = accs[msg.sender];
13     acc_mem.coins =  acc_mem.coins - value;
14     accs[msg.sender] = acc_mem;
15     msg.sender.transfer(value);
16   }
17 }
```

Figure 1: Wallet contract example.

and a volatile part called the *memory*. The behaviour of variable allocation and initialisation, and assignment between variables, depend on which of these hold the relevant variables. Ethereum blockchain stores a collection of instances of different contracts. Contracts can be dynamically created, and they can interact with one another. Typically, model-checking frameworks focus on a specific contract instance, but they create some abstraction to capture the behaviour and state of the other contracts too. This abstraction is required, for instance, to capture inter-contract calls. A verification harness drives the behaviour of the blockchain and of the specific instance being analysed. A harness would allow the execution of sequences of calls of this instance's public functions. It has to account, however, for the possibility of functions of other contracts being also executed, or even new contracts being created, in between these calls. Precisely capturing the behaviour of all the other contracts is infeasible given how numerous they may be and that each can be an arbitrary Solidity program. The harness should be as precise as it can be while making contract verification tractable.

Contributions. We propose a new formalisation of Solidity using Boogie [15, 30], an *intermediate verification language*, that addresses our three challenges. For (1), we propose a new way to represent the state of the blockchain, which depicts the storage associated with each contract instance, as well as the memory used to execute contracts. Challenge (2) is addressed by a technique that we proposed called *lazy blockchain exploration*. It creates contracts on-the-fly as they are need in the analysis. As for (3), we propose two *memory-precise verification harnesses*. While our *contract harness* zero-initialises and precisely executes a specific contract, the *function harness* non-deterministically initialises the contract and executes one of its functions. The function harness allows errors deep in the execution of a contract to be quickly found. A key property of our Boogie encoding is that blockchain exploration (i.e. smart-contract execution) always manipulates a well-formed state, namely, variables are well-typed, properly allocated, and correctly structured in memory and storage. Other frameworks do not offer this guarantee and so are in danger of identifying unrealistic errors. Moreover, we also introduce *Solidifier*: a bounded model checker for Solidity that uses Corral [29], a bounded model checker for Boogie, to analyse our encoding. It interprets errors and counterexamples of Corral back into Solidity. Solidifier has been designed to find bugs, i.e. behaviours falsifying properties, which should be then fixed by developers and re-verified. Moreover, our evaluation seems to suggest that not only does Solidifier (and our Boogie encoding) more faithfully capture Solidity's semantics than similar tools, but it also does so with an arguably insignificant verification speed overhead.

Outline. Section 2 briefly introduces key concepts of Ethereum, Solidity and Boogie. Section 3 describes our Boogie encoding for Solidity and Ethereum, focusing on introducing our memory modelling, lazy blockchain exploration, and memory-precise verification harnesses. We discuss and evaluate Solidifier against other tools in Section 4. We present related work in Section 5 and our conclusions in Section 6.

2 BACKGROUND

The concept of a blockchain was introduced to prevent double spending in the Bitcoin network [35]. Since then it has been generalised in many ways, including the proposal of smart contracts as a flexible mechanism to extend its capabilities [2, 11]. A blockchain is a transaction-based system designed with a fixed set of rules that defines its basic transaction logic. These rules are enforced by a consensus protocol executed by the blockchain nodes; reaching a consensus is, to a large extent, what prevents nodes from misbehaving and, consequently, ensures the correct behaviour of a blockchain. The blockchain's behaviour emerges from the processing of transactions and the effects it has on the underlying database. Once the blockchain network is deployed these rules are fixed and cannot be modified. One way to make the behaviour of a blockchain more flexible is to allow for transactions that trigger the execution of some function - such transactions are valid if the underlying computation completes without raising errors. This mechanism is precisely what smart-contract-based blockchains implement.

The Ethereum blockchain (or just Ethereum, for short) is arguably the most widely-known and used smart-contract-based blockchain [2]. Its database associates *addresses* with *address cells*. Each address has (stored in its corresponding address cell) an associated balance of crypto coins and (maybe) some smart-contract code and persistent state. An address in Ethereum is either unused, a simple currency-holding address (which we call a *simple address*), or a smart-contract instance. While a simple address has its balance managed by a secret cryptographic key, a contract address' assets are managed by its code. The code of an Ethereum smart contract is ultimately represented using EVM *bytecode* (and executed by the EVM) but, typically, smart contracts are created using the high-level language Solidity [6] and later compiled into bytecode. An Ethereum transaction can be used to *deploy* a smart contract to (i.e. create a smart-contract instance at) a

blockchain *address*. Once deployed, smart contract functions can be invoked using a contract-execution transaction by specifying what address is to be called and what function needs to be executed. We focus on Solidity as it is apparently the most popular language for creating smart contract.

A *Solidity program* is composed by a set of *smart contract* declarations. A contract is similar to a class in object-oriented languages. It can declare a set of *user-defined types*, a set of *member* (*state*) *variables*, and a set of *member functions*. While the member variables capture the (persistent) state of the contract, the functions describe its behaviour. Functions have visibility modifiers to specify whether they are part of the public interface of the contract. For instance, the Solidity program in Fig. 1, which we use as a running example, has a single contract `Wallet` that behaves like a toy bank: addresses can open and close an account, and deposit and withdraw money from their accounts; `withdraw` is the only function that is relevant to our exposition. For brevity, we only introduce the Solidity constructs that are relevant for our exposition - see [6] for a full account of the language.

Like traditional programs, smart contracts are also prone to flaws. Unlike most traditional programs, however, smart contracts manage digital assets that can be exposed by such flaws. For address `msg.sender`, the function `withdraw` in Fig. 1 updates the `Account` associated to it by decrementing its balance of coins by `value`. Solidity functions have an implicit input parameter `msg.sender` that gives the address of caller. Without the `require` statement in Line 10, the subtraction `acc_mem[msg.sender].coins - value` in Line 13 could underflow causing `accs[msg.sender]` to hold a balance of coins much larger than it should be. Honouring this balance would represent a catastrophic loss for `Wallet`.

We formalise the behaviour of Ethereum and Solidity using Boogie [30]: a simple but powerful intermediate verification language. A Boogie program declares global variables and procedures. The procedures declare some local variables and a list of statements describing its behaviour. For instance, Fig. 3 depicts the Boogie procedure encoding the `Wallet` contract's constructor; we describe in detail this procedure in the next section. Boogie supports traditional commands such as while-loop, if-then-else, assignment and procedure call, and also specification-based statements. While `assume` causes the program to fail silently if its condition is not met, `assert` generates an error; `havoc` non-deterministically assigns a new value to its expression. To simplify our presentation, we extend Boogie to support the `Ref` type, and both enumeration and record declaration; they can be easily specified using Boogie's built-in constructs. Type `Ref` contains (infinite) special values $\texttt{Ref}_i$ that can be tested for equality. Type $\texttt{[}Ty\texttt{]}\ Ty'$ denotes the set of total mappings from Ty to Ty'. As expressions, our extended Boogie has left-value expressions: identifier, mapping access, record access; integer arithmetic and boolean expressions; quantified expressions; etc. Roughly speaking, we use Boogie's global variables to model the execution state of Ethereum and procedures to encode the behaviour of Solidity smart contracts.

3 FORMALISING SOLIDITY AND THE ETHEREUM BLOCKCHAIN

We capture the state of Ethereum addresses using global variable `s : [Address] AddressCell`, where address cell `s[addr]` stores the state of address `addr` $\in$ `Address`. `Address` represents the set of 160-bit unsigned integers. We represent fixed-sized integers using Boogie's `int`,[1] and arithmetic expressions on these integers are encoded using Boogie's operations on mathematical integers with the addition of wrap-around behaviour to capture overflow or underflow. This choice is based on the findings in [23]. `Address` is the Boogie counterpart of Solidity's `address`. An address cell is a record with 4 elements `type` $\in$ `AddressType`, `balance` $\in$ `UInt` (where `UInt` gives the set of 256-bit unsigned integers), `members`, and `storage` - the latter two elements record the state of smart contracts, which we discuss next. `AddressType` is an enumeration with values `Unused`, `SimpleAddress`, and contract names, representing unused, simple, and contract-instance addresses, respectively. `UInt` (`Int`) is the Boogie counterpart of Solidity's `uint` (`int`).

Solidity functions have an implicit `this` variable that gives the contract instance on which the function is applied. Each Solidity function is encoded as a Boogie procedure with address `this` and address `msg.sender` as input parameters. The built-in address function `addr.transfer(v)` transfers money (`v`) from address `this` to `addr`. Command `require(`*cond*`)` raises an exception if its *cond* is false. Exceptions cause the current call to be reverted, and they are propagated upwards in the call stack.[2] Solidity has also traditional commands such as while and for loops, if-then-else construct, assignment and function call. The function `transfer` is encoded as a Boogie procedure that carries out the appropriate funds transfer by manipulating the corresponding addresses' balances, whereas `require(`*cond*`)` is encoded as `assume bcond`, where `bcond` is the Boogie translation of *cond*. Solidity's while and for loops, and if-then-else constructs are captured with Boogie's counterpart while loops and if-then-elses.

We use Solidity library `Verification` to introduce verification functions `Assume(bool b)` and `Assert(bool b)`. It simply brings the corresponding Boogie primitives into Solidity. While `Verification.Assume(`*cond*`)` ensures that *cond* holds at the point in the program in which it appears, expression `Verification.Assert(`*cond*`)` raises an error if *cond* is false.

3.1 Solidity's memory model

Our formalisation supports the following basic Solidity types: addresses (denoted in Solidity by `address`), contracts (denoted by contract names), signed and unsigned 256-bit integers (denoted by `int` and `uint`, respectively), and boolean (`bool`). Both address and contract types are 160-bit unsigned integers that identify an address in the blockchain. A contract type, however, is an identifier annotated/typed with the type of the contract that it stores. In addition to these basic types, we support arrays (denoted by `MemberType[]`), mappings (denoted

[1] We use `where` clauses to bound the `int`'s its size.
[2] This propagation can be stopped by command `call`.

by `Domain => Range`), and user-defined types (these can be `enum`s or `struct`s). Recursive types are not supported.

The element `storage` $\in$ `[Ref] RefCell` (which we call a *reference mapping*) of an address cell stores the values of state variable for the smart-contract instance at this address. Reference cell `storage[r]` gives the *reference cell* that reference r points to. A reference cell is a record with two elements: `type` $\in$ `RefCellType` stores the Solidity type of the value stored or the special value `None`, and `value` $\in$ `RefCellValue` the actual value stored. While type `None` denotes that the reference has not been allocated yet, a Solidity type tags the reference cell with the type of the value stored. We represent this Solidity-type tag with a Boogie enumeration that lists all types used in the Solidity program we are encoding. `RefCellValue` gives the disjoint union of all the types that can be stored in a reference cell for the Solidity program being encoded. The element `members` $\in$ `[MemberIds] Ref` captures where member variable values are stored. For member name `m` $\in$ `MemberIds`, `members[m]` gives the reference where `m` is stored in `storage`.

In our formalisation, we use a value in $\mathit{RefType}(T)$ to capture the state of a Solidity variable of type T stored in a reference mapping. The state of a variable of type `uint`, `int`, or `bool` is captured using Boogie types `UInt`, `Int`, or `bool`, respectively. The state of a variable of type `address` or contract is represented by `Address`. Enum definitions in a Solidity program give rise to a corresponding Boogie enumeration definition, and their state is represented by a value of this corresponding enumeration. $\mathit{RefType}(T\,\texttt{[]})$ gives a record with members `length` $\in$ `UInt` and `data` $\in$ `[UInt]` T', if T is a reference type (i.e. T is a Solidity array or struct) then T' is `Ref`, otherwise T' is $\mathit{RefType}(T)$. In general, reference-type elements of a composite type are represented by references that point to reference cells storing the members' state, whereas non-reference-type elements are simply represented by their values. So, a struct is represented by a record where each member m of type T gives rise to a member m of type `Ref` if T is a reference type, and $\mathit{RefType}(T)$ otherwise. Mapping D `=>` R is represented by a Boogie total mapping `[`$\mathit{RefType}(D)$`]` R' where R' is `Ref` or $\mathit{RefType}(R)$ according to whether or not R is a reference type. The domain type of this mapping is given by $\mathit{RefType}(D)$ as we only allow Solidity mappings to have a basic domain type.

For instance, Fig. 2 depicts a possible address cell configuration for an instance of our `Wallet` contract. While $\texttt{Ref}_1$ keeps track of the `accs` variable, $\texttt{Ref}_2$ and $\texttt{Ref}_3$ keep track of mapped `Account` structs.

A contract also has at its disposal a block of volatile memory spaces, called the *memory*, that can store reference-type (i.e. structs and arrays) values on demand as the contract needs more resources to execute; these memory spaces are disjoint from `storage`'s. We capture the memory with Boogie global variable `memory` $\in$ `[Ref] RefCell`. Each blockchain transaction is processed with a fresh memory.

We capture a Solidity local variable (parameter) of type T by a Boogie local variable (parameter) of type $\mathit{LocalType}(T)$. For a reference type T, $\mathit{LocalType}(T) =$ `Ref` and the reference value points to a value in $\mathit{RefType}(T)$, whereas, for a value type T, $\mathit{LocalType}(T) = \mathit{RefType}(T)$. These references will point to an appropriate cell in memory or storage depending on their type. In Solidity, variables are initialised to their default values. Addresses, contracts, and integers have the integer literal zero as their default values, while *false* is the boolean default value. The default value for composite types is created by having its members set to their default values. Mutable-sized arrays are initialised to the empty array, whereas fixed-sized arrays have a constant size - their data elements are set to their default values.

- `type` $=$ `Wallet`;
- `balance` $= 210$;
- `members` $=$ { `accs` $\rightarrow \texttt{Ref}_1$ };
- `storage` $= \{\texttt{Ref}_1 \rightarrow \{$`type` : `address => Account`, `value` : $\{0 \rightarrow \texttt{Ref}_2, 1 \rightarrow \texttt{Ref}_3, \ldots\}\}$, $\texttt{Ref}_2 \rightarrow \{$`type` : `Account`, `value` : $\{$`coins` : 10, `status` : `Open`$\}\}$ $\texttt{Ref}_3 \rightarrow \{$`type` : `Account`, `value` : $\{$`coins` : 0, `status` : `Closed`$\}\}, \ldots\}$

Figure 2: Address cell example for `Wallet`.

3.1.1 Variable allocation and initialisation. State variables and memory pointers also have their values properly allocated in storage and memory, respectively. We use Boogie code template $\mathit{Allocate}(rm, ref, T)$, given as follows, to allocate a single reference cell, where rm is a reference map, ref a reference expression, and T a Solidity type.

```
assume rm[ref].type == None;
rm[ref].type := T;
```

The template $\mathit{AllocateMany}(rm, ref, T)$ can allocate multiple references at once. For instance, to allocate all the `Account` references pointed to by a `address => Account` map, *AllocateMany* is as follows. *ref* and *rm* could be, for instance, $\texttt{Ref}_1$ and `storage`, respectively, as per Fig. 2. We use *map* as a shorthand for *rm*`[`*ref*`].value`; it maps addresses to references.

```
1  assume ∀ a : Address • rm[map[a]].type == None;
2  assume ∀ a,b : Address • a != b ⇒ map[a] != map[b
       ];
3  prm := rm;
4  havoc rm;
5  assume ∀ r : Ref • prm[r].type != None ⇒ prm[r]
       == rm[r];
6  assume ∀ a : Address • rm[map[a]].type = Account;
```

Line 1 ensures that the new references we want to allocate point to unallocated reference cells, whereas Line 2 ensures these new references are not aliased. The *prm* is a variable that captures the value of the reference mapping *rm* (in Line 3) before it is `havoc`ed in Line 4; each Boogie procedure in our encoding declares an auxiliary variable `preRefMap : [Ref] RefCell` for this purpose. Line 5 uses the *prm* to ensure that the new `havoc`ed value of *rm* preserves (pre-`havoc`) allocated reference cells. Line 6 *de-facto* allocates the new references. *AllocateMany* works similarly for other dynamically-sized types. Our encoding relies on a similar

```
1  procedure Wallet_constructor() {
2    var preRefMap : [Ref] RefCell;
3    var this : Address;
4
5    s[this].type := Wallet;
6    s[this].balance := 0;
7    Allocate(s[this].storage, s[this].member[accs],
         address => Account)
8    AllocateMany(s[this].storage, s[this].member[accs
         ], address => Account)
9    Initialise(s[this].storage, s[this].member[accs],
         address => Account)
10   InitialiseMany(s[this].storage, s[this].member[
         accs], address => Account)
11 }
```

Figure 3: Boogie procedure for `Wallet` constructor.

strategy to initialise variables to their default values using counterpart functions *Initialise* and *InitialiseMany*.

Lines 7-10 in Fig. 3 depict how we capture in Boogie the allocation and initialisation of member variable `accs` by the implicit constructor of the `Wallet` contract. While *Allocate* allocates the reference storing variable `accs`, *AllocationMany* allocates all the references `accs[a]` where `a` is an address. Initialisation occurs similarly. In Solidity, a contract that does not define a constructor is given one that zero initialises its storage and balance. The constructor procedure has `this` as a local variable instead of an input parameter like the other encoded procedures.

State variables are allocated and initialised when the contract is created and memory variables are allocated and initialised when a memory pointer is declared. Solidity also allows the creation of memory arrays of T with a fixed runtime-computed size n using construct `new` T`[](`n`)`. The declaration of a memory pointer to an array will trigger the allocation of an empty array that cannot be enlarged. Hence, the need for this construct. Unlike memory arrays, storage ones can have their size dynamically altered. When an array is shrunk, members no longer in range are reset to their default values. We encode this resetting behaviour using Boogie's while loop.

3.1.2 Assignment. A Solidity reference-type value is represented by either a *storage reference*, a *storage pointer* or a *memory pointer*. Local variables and parameters of a reference type are declared with a *data location* modifier that states whether they denote a memory or storage pointer. On the other hand, referencing state variables, accessing their members, or accessing members of a storage pointer give rise to storage-reference expressions. For instance, in Fig. 1, Line 11, memory pointer to `acc_mem` is created, whereas expression `accs[msg.sender]` in Line 14 denotes a storage reference.

In Solidity, while an assignment between value-type expressions results in a simple copy, one between reference-type expressions can result in *deep copying*, depending on where the values of these expressions are stored. Assignments are used, for instance, to "save" some transient state in memory into the persistent storage of the contract. Table 1 summarises the semantics of Solidity assignments involving reference-type

Left-hand side	Right-hand side	Semantics
Storage reference	Any type	Deep copy
Memory pointer	Storage reference	New deep copy
	Storage pointer	New deep copy
	Memory pointer	Shallow copy (Aliasing)
Storage pointer	Storage reference	Shallow copy (Aliasing)
	Storage pointer	Shallow copy (Aliasing)
	Memory pointer	Disallowed by compiler

Table 1: Assignment semantics for reference types.

expressions. If the left-hand side of the assignment is a memory pointer, a deep copy is created on a *newly allocated* reference tree. On the other hand, deep copying into a storage reference does not create a newly allocated reference - the existing already-allocated reference tree is used. For instance, in Fig. 1, the assignment in Line 12 has a left-hand-side memory pointer and a storage-reference as right-hand side. So, it creates a new copy of the storage value `accs[msg.sender]` in memory for which pointer is stored in variable `acc_mem`. On the other hand, the assignment in Line 14 deep copies the memory value pointed by `acc_mem` into the storage reference tree pointed by `accs[msg.sender]`. It should be noted that an array deep copying might trigger a resizing of the left-hand-side expression; elements which become out of bounds are reset. We take that into account in our encoding.

Solidity has an unusual but deterministic mechanism to "allocate" storage references that makes it rather different from traditional languages. It uses a *cryptographic hash function* to decide/locate which reference cells are used to store contract state variables. Roughly speaking, the member-accessing expression on a contract state variable is used as an input to this hash function and it outputs the location of the reference cell in storage where this member is stored - in our encoding, we abstract these concrete locations using `Ref`. As a consequence, a contract variable and its members are always stored in the same reference cell, which leads to unintuitive/unusual aliasings between storage pointers and storage references. For instance, let us assume that just before Line 11 in Fig. 1, we had the storage pointer declaration `Account storage acc_stor = accs[msg.sender]`. In Line 14, the memory value pointed by `acc_mem` is deep copied into `accs[msg.sender]` but, even after Line 14, `acc_stor` and `accs[msg.sender]` are still aliased. In many other languages, however, Line 14 would make `accs[msg.sender]` point to a new reference cell - and turn `acc_stor` into a dangling/stale pointer. We capture this unusual behaviour by having storage references completely allocated when contracts are created and making sure that assignments do not cause the allocation of new `storage` reference cells as per Table 1. In Solidity, accessing different contract state variable members could lead to the same hash (and location) but this sort of *collision* is assumed not to happen in practice. Hence, in our encoding, we make sure that distinct contract state variables and their members are allocated in distinct reference cells - this assumption is also made in other work [22, 23, 45].

```
1  procedure Wallet_withdraw(this : Address, msg.
       sender : Address, value : UInt) {
2  var preRefMap : [Ref] RefCell;
3  var acc_mem : Ref; // Line 11
4
5  if (s[this].type == Unused){
6    s[this].type := Wallet;
7    Allocate(s[this].storage, s[this].member[accs],
         address => Account)
8    AllocateMany(s[this].storage, s[this].member[
         accs], address => Account)
9  }
10 assume s[this].type == Wallet;
11
12 assume s[this].storage[s[this].storage[s[this].
       members[accs]].value[msg.sender]].status ==
       Status.Open; // Line 9
13 assume s[this].storage[s[this].storage[s[this].
       members[accs]].value[msg.sender]].coins >=
       value; // Line 10
14 Allocate(memory, acc_mem, Account) // Line 11
15 Initialise(memory, acc_mem, Account) // Line 11
16 Allocate(memory, acc_mem, Account) // Line 12
17 memory[acc_mem] := s[this].storage[s[this].
       storage[s[this].members[accs]].value[msg.
       sender]]; // Line 12
18 memory[acc_mem].value.coins := memory[acc_mem].
       value.coins - msg.value; // Line 13
19 s[this].storage[s[this].storage[s[this].members[
       accs]].value[msg.sender]] := memory[acc_mem
       ]; // Line 14
20 call transfer(this,msg.sender,value); // Line 15
21 }
```

Figure 4: Boogie procedure for `Wallet withdraw`.

3.2 Contract deployment and function call

An instance of contract C can be created through a blockchain transaction or a call to Solidity's command `new` C - these are captured in our encoding as a call to C's constructor procedure. We add the Boogie code template denoted by *DeployContract*(C) to the beginning of the Boogie procedure capturing C's constructor. It creates and initialises a new C instance in the blockchain. In Fig. 3, Lines 5-10 correspond *DeployContract*(`Wallet`).

A Solidity contract function call is translated into a Boogie call of the corresponding procedure. Our Boogie encoding proposes *lazy contract deployment*. A function call might target a yet uninitialised address. In this case, before the function is executed, the appropriate contract is deployed at this address. We use the Boogie code template *LazyContractDeployment*(C) to capture lazy deployment. Fig. 4 presents a simplified version of the encoding of `Wallet withdraw`. Lines 5-9 depict *LazyContractDeployment*(`Wallet`), Lines 12-20 are annotated with comments pointing to the line in Fig. 1 they are translated from.

Lazy contract deployment ensures that as the verifier explores Ethereum executions, contracts have properly-allocated state variables - this guarantees, for instance, that state variables are not aliased. Note that *LazyContractDeployment*(C) deploys an instance of C where the values of member variables and its balance are non-deterministically initialised, whereas *DeployContract*(C) creates an instance with their default values. Procedures also have a condition that ensures that memory references are well typed.

We restrict Solidity's semantics when encoding the behaviour of a contract function call. Our encoding uses contract types to validate the expected type of contracts being called. Solidity, however, executes a function of the called contract that matches the expected signature, even if the type of the contract being called is completely different from what the caller expects. Hence, a function with the same signature but with a completely different behaviour can be executed. Even worse than that, in Solidity, if no function matches a given signature, the fallback function deployed in the address called is executed. To avoid this sort of behaviour, each procedure capturing a function of contract C has validation `assume s[this].type == C` before its actual behaviour is executed; Line 10 in Fig. 4 represents such a contract type validation. This assumption should make verification simpler as our encoding/formalisation does not have to assume that any function of any type of contract can be triggered. A similar assumption is made by VeriSol [45].

3.3 Memory-precise verification harnesses

We use a *verification harness* to capture relevant executions of the blockchain and search for smart contract errors. We propose a *contract harness* and a *function harness*. They define a `main` procedure, and a `callP` procedure that captures Solidity's low-level `call` primitive. The expression `addr.call(`*sign*`)` calls a function with signature *sign* for the contract instance at address `addr`. Our harnesses use a controlled non-deterministic reinitialisation of `storage` and `memory` to abstract that a sequence of blockchain transactions were executed. They are *memory precise* in the sense that this reinitialisation preserves well-formedness for `storage` and `memory`, namely, values stored are well typed, properly allocated, and correctly structured.

For a contract of type C, our contract harness analyses the behaviour of an instance of C at some non-deterministically chosen address. This analysis does not assume anything about other addresses apart from the fact that their storage and memory have been correctly initialised and are well typed - this is achieved thanks to the lazy deployment feature of our encoding. The `main` procedure creates an instance of C at the address given by global variable `main_contract` and executes an arbitrary sequence of interface functions. This procedure ensures that the initial non-deterministically-created reference cells of `memory` are well typed and that initially `storage` records each address as either unitialised or a simple address. For each new interface function call (the constructor is not an interface function), it non-deterministically chooses a function and initialises its arguments. For each new call, it also `havocs` the storage and balance of all addresses except for `main_contract`: basic values can change but references and reference-cell types are fixed. These `havocings` conservatively simulate the execution of transactions to move funds and

trigger functions for other addresses. The `call` primitive is treated as an external/unknown function call. The procedure `callP` works similarly to `main`. It can execute any (finite) sequence of interface functions of C on `main_contract` and it `havoc`s the storage and balance of other addresses. This simulates the execution of an external function that can call back into a function of C, i.e. exhibit some reentrant behaviour.

Since this harness precisely (up to abstractions) deploys and executes the contract C at address `main_contract`, the interface functions calls are executed from valid/reachable states of C, namely, states satisfying C's invariants.

Our function harness executes a single call to a chosen function f of C. It analyses the behaviour of f when execution from a non-deterministically initialised state of a C's instance. This analysis also does not assume anything about other addresses apart from the fact that their storage and memory have been correctly initialised and are well typed, as per lazy deployment. The `main` procedure simply calls the procedure corresponding to f, whereas `callP` `havoc`s storage, memory, and balance (in a way that preserves the well-formedness of storages and memory) of all addresses to denote that some arbitrary execution took place.

Unlike the contract harness, the function harness can execute f from a state that is not reachable by a well-behaved instance of C, thus possibly reaching actually unreachable errors. However, this harness can find deep errors that might not be reachable by our contract harness, given a fixed bound. While the contract harness examine bounded executions starting in the contract's initial state, the function harness examines bounded executions starting from any state.

4 SOLIDIFIER

Solidifier carries out a bounded verification of Solidity programs looking for failing assertions - our Boogie encoding is checked by Corral [29], a *reachability-modulo-theories verifier*. For an input system, traditional bounded model checkers create a symbolic constraint to explore the system states reached with at most k transitions from its initial state. Corral, on the other hand, uses counterexample-guided abstraction refinement (CEGAR) to (possibly) simplify and incrementally search this bounded state space. This abstraction process leads to an intermediate over-approximation of the input program that might even be sufficient to *prove* properties *irrespective of the bound* k set. Its intended use, however, is to find property violations.

Solidifier runs the Solidity compiler [5] as an auxiliary tool to validate the input program and to generate its typed abstract syntax tree, later parsed by Solidifier. The user of Solidifier chooses between a contract or function harness. Solidifier also interprets Corral results and information back into Solidity. Aside from the verification primitives, functions prefixed with `CexPrint_` and with a single basic-type parameter can be declared in the `Verification` library. When called, they inform Solidifier to print the argument's value at that point in a counterexample, if one reaching this call exists. Solidifier outputs either: a counterexample trace leading to a failing assertion, that no failing assertion can be ever reached (irrespective of the bound), that no failling assertion could be reached for the predetermined bound, or it might fail to produce any output if the underlying solver is unable to solve the constraints generated by Corral.[3]

4.1 Evaluation

We illustrate the capabilities of Solidifier by comparing it against solc-verify [22], VeriSol [7] and Mythril [34] in the analysis of 23 contracts. In [4], we provide instructions to build the Docker container that we used for our evalution. It should provide a convenient way to reproduce our results. The container contains the Solidifier binary, the Solidity examples we used, and the tools we compared Solidifier against. We conducted our evaluation on a MacBookPro with Intel(R) Core(TM) i7-8559U CPU @ 2.70GHz and 16GB of RAM, and with Docker Engine 18.09.1. Table 2 depicts our results.[4]

Comparing checkers for Solidity is not an easy task given how quickly the language evolves. Between September 2018 and July 2020, Solidity has gone through 3 backward-incompatible language updates - from Solidity 0.4.* to 0.7.*. Our encoding should be compatible with a core language that has been reasonably stable across this evolution (at least 0.4.* - 0.6.*) but some features have been discontinued. For instance, since 0.6.* explicit dynamic-array resizing has been forbidden by the compiler. However, contracts of previous versions can still be (and still are) deployed to Ethereum. Different tools handle different subsets of Solidity and different versions.

We chose to compare our tool against VeriSol and solc-verify because they are reasonably mature and complete, they reason about contracts at the Solidity-level, and they are Boogie-based - albeit, using a different encodings and, in solc-verify's case, a different Boogie backend, the Boogie Verifier [15]. While the former is a bounded model checker, the latter is an unbounded one. We add Mythril to our comparison as it is the archetypical tool that uses symbolic execution to analyse a contract's EVM bytecode - this comparison should give a sense of how our tool compares with EVM-based tools.

Our evaluation uses Azure blockchain workbench samples [1] - which has been used by other papers as a benchmark too [22, 23, 45] -, Solidity documentation examples [6] (OpenAuction, OpenAuctionWithCall, and Voting), and some examples created by us. Unlike the situation with tools that look for a specific behaviour pattern, we cannot simply scour the Ethereum blockchain looking for vast numbers of contracts to analyse because in many cases only the EVM bytecode is available and even when Solidity code is made available, there is no documentation that we could rely upon to specify what property the developer of the contract intended to guarantee.

[3] The use of undecidable theories such as non-linear arithmetic, which we use to capture the modulo and multiplication operation, might lead the underlying solver to an inconclusive result.

[4] The version of VeriSol used here is not the same as the one used in [45], neither are the properties that we check here and the ones checked there - they should be similar but note that VeriSol checks for a notion of compliance to the Azure workflow policies.

We choose these examples because they were either developed by us or accompanied by informal textual specification describing what they should achieve.

For each program we give the time taken in seconds to analyse the contract with the outcome in parenthesis: $n\times$ means that n violations were found (just $\times$ denotes a single violation), ✓ that the properties have been proved, *bd* represents that no violation has been found in the portion of behaviour analysed (within a transition bound, for instance), *Error* reports a tool's internal error, * represents a timeout in the analysis of the contract, *Bug* reports a wrong violation due to a bug in Mythril for RoomThermostat. VeriSol and Solidifier report as soon as they find a violation, solc-verify check all functions possibly finding more than one violation per function, and Mythril explores the contract up to a certain coverage criterion possibly finding multiple violations. Aside from assertion violations, the analysis mode under which we run Mythril also looks for out-of-bound array accesses. We set a 300 seconds timeout for analysing each program. The cells with a grey background present a wrong output: they either state that a violation exists when it does not or that a property holds when it does not. We refer to *precision* as the number of wrong outcomes: the more precise the tool is the fewer wrong outcomes it provides. Solidifier and VeriSol use Corral with a fixed recursion bound of 128.

The Azure blockchain workbench samples' specification describe high-level transitions for contract states, which we capture with assertions. We proceeded to fix the contracts that did not meet their corresponding specification originally. The properties specified for these contracts mainly restrict pre- and post-states of functions. Corral's lazy-inlining over-approximation [29] is usually sufficient to prove such properties irrespective of bound. For Stater in PingPongGame, however, proving the associated property requires deriving a non-trivial invariant for a recursion.

For OpenAuction(WithCall) we capture the correctness of refunding someone's bid, whereas for Voting we capture the invariant that the number of votes initially available is equal to the sum of votes still to be cast plus votes already cast throughout the election process. For Aliasing and StorageDeterministicLayout, we use our function harness to analyse function t. While Aliasing captures properties arising from Solidity's memory model and its different aliasing/deep-copy possibilities, StorageDeterministicLayout captures properties arising from the deterministic organisation of member variables in storage. Finally, all the versions of Wallet, based on Fig. 1, intend to capture (code and properties of) a typical use of smart contracts to tokenise digital assets, added with extra memory model intricacies.

These results are entirely consistent with how these different tools operate. As an EVM-bytecode checker, Mythril should be the most semantically-faithful tool but this faithfulness comes at the expense of speed: it should not report any imprecise result but its analyses take much longer than the others. solc-verify's use of a modular (assume-guarantee) verifier intrinsically over-approximates the behaviour of contract functions. Without user-input annotations to more precisely capture the behaviour of a contract, its functions are generally analysed as if they were executed from an arbitrary contract state. Hence, many false errors are reported. Solidifier significantly outperforms VeriSol in terms of precision and speed. We attribute their difference in precision and speed to different Boogie encodings of Solidity as well as to distinct configurations of Corral.

Azure workbench original				
Example	Solidifier	solc-verify	VeriSol	Mythril
AssetTransfer	2.89 (×)	1.79 (×)	3.01 (×)	53.55 (*bd*)
BasicProvenance	0.87 (×)	1.42 (2×)	2.03 (×)	8.36 (×)
ItemListingBazaar	2.01 (×)	1.48 (3×)	4.42 (×)	78.58 (3×)
DigitalLocker	1.23 (×)	1.52 (9×)	2.20 (×)	36.20 (8×)
DefectiveComponentC	7.42 (×)	1.50 (×)	5.90 (×)	13.05 (×)
FrequentFlyerRC	0.88 (✓)	1.53 (×)	*	*
HelloBlockchain	0.99 (×)	1.40 (2×)	1.79 (×)	6.22 (2×)
PingPongGame	1.88 (×)	1.48 (2×)	3.88 (×)	111.54 (×)
RefrigeratedTrans	1.66 (×)	1.45 (3×)	2.23 (×)	71.05 (×)
RefrigeratedTransTime	1.72 (×)	1.44 (3×)	2.27 (×)	90.97 (×)
RoomThermostat	0.77 (✓)	1.39 (✓)	1.81 (×)	*Bug*
SimpleMarketplace	1.06 (×)	1.43 (×)	1.97 (×)	9.01 (×)

Azure workbench fixed				
Example	Solidifier	solc-verify	VeriSol	Mythril
AssetTransfer	1.58 (✓)	1.49 (✓)	2.44 (*bd*)	54.91 (*bd*)
BasicProvenance	0.75 (✓)	1.41 (2×)	*	7.50 (*bd*)
ItemListingBazaar	2.63 (✓)	1.51 (×)	*	72.91 (*bd*)
DigitalLocker	1.15 (✓)	1.51 (×)	*	19.49 (*bd*)
DefectiveComponentC	0.84 (✓)	1.50 (×)	*	12.42 (*bd*)
HelloBlockchain	0.71 (✓)	1.36 (✓)	1.85 (*bd*)	4.88 (*bd*)
PingPongGame	*	1.50 (×)	34.61 (*bd*)	108.13 (*bd*)
RefrigeratedTrans	2.11 (✓)	1.47 (3×)	*	69.67 (*bd*)
RefrigeratedTransTime	2.09 (✓)	1.47 (3×)	*	89.45 (*bd*)
SimpleMarketplace	1.71 (✓)	1.43 (×)	*	8.55 (*bd*)

Others examples				
Example	Solidifier	solc-verify	VeriSol	Mythril
Aliasing	1.87 (✓)	1.43 (✓)	1.99 (×)	5.52 (*bd*)
StorageDeterministic	1.90 (✓)	1.43 (×)	1.98 (×)	10.01(*bd*)
OpenAuction	*	1.45 (✓)	2.25 (*bd*)	*Error*
OpenAuctionWithCall	3.34 (×)	1.54 (3×)	*Error*	*Error*
Voting	*	1.58 (×)	2.33 (×)	*
Wallet	3.74 (×)	*Error*	2.09 (×)	37.02 (*bd*)
WalletNoOver	*	*Error*	2.12 (×)	29.49 (*bd*)
WalletNoOverCall	3.97 (×)	*Error*	*Error*	42.98 (*bd*)
WalletNoOverCallLocking	*	*Error*	*Error*	57.93 (*bd*)

Table 2: Evaluation results.

5 RELATED WORK

The semantics of the EVM has been formalised in traditional interactive theorem provers [10, 25, 36]. The work in [24] formalises the behaviour of the EVM using the $\mathbb{K}$ framework [38], whereas the authors of [21] use F* [40] to this end. These frameworks are designed to prove properties of systems as opposed to falsifying them. They are very precise and scalable (as they can establish properties of a complex system in a reasonable time) but they have a low degree of automation and require a considerable amount of manual effort. They are as precise and scalable as the proof engineer that manually guides the proving effort.

The following tools automatically examine an approximation of the behaviour of a smart contract to look for specific

behavioural/code patterns that tend to lead to vulnerabilities. *SmartCheck* [43] translates Solidity code into a XML-based intermediate language that is later queried for specific syntactic patterns. *Oyente* [32], like *Mythril* [34], use symbolic execution to examine an under-approximation of some EVM bytecode; Oyente, in particular, can be fairly ineffective as far as code-coverage is concerned [44]. *EtherTrust* [20] and *Securify* [44] are based on the generation of some symbolic constraints that soundly but coarsely over-approximate the behaviour of EVM bytecode; while EtherTrust uses horn clauses to represent these constraints, Securify uses a datalog program. They are intended to prove properties of programs rather than falsify them.

VerX [37] is a tool that uses symbolic execution and delayed predicate abstraction to check temporal (safety) properties of smart contracts. It is an automated tool that allows users to specify, using temporal logic [33], functional properties of smart contracts [39]; sometimes manual input is required to construct counterexamples.

VeriSol [45] and *solc-verify* [22, 23] are verifiers based on a Boogie encoding of Solidity. Our approach significantly differs from theirs. The *solc-verify* verifier proposes an *unbounded* model checker for Solidity. Their Boogie encoding is checked by the Boogie verifier [15]. Given a Solidity program annotated with pre- and post-conditions, contract and loop invariants, and assertions, the verifier tries to discharge the proof obligations derived from the specification. This sort of modular/assume-guarantee reasoning can be quite precise and scalable but that intrinsically depends on the ingenuity of the specifier. The new version of solc-verify (in [22]) relies on a precise memory model that is similar to ours. They have slight differences such as ours uses `RefCell`'s type to allocate new memory cells whereas solc-verify's uses a reference count for that. There is, however, a significant difference between our encoding and solc-verify's that is a consequence of their distinct verification methodologies. A modular verifier checks functions individually, that is, it conservatively approximates the behaviour of function calls by their specification as opposed to precisely analysing the behaviour of called function. As solc-verify expects users to manually annotate contracts with specifications, function calls are modelled as precisely as users' manual annotations. Their approach does not propose a precise way to automatically capture the behaviour of function calls. We propose the use of bounded model checking instead. This approach tends to be less scalable since functions calls are precisely analysed as they are explored up to a certain execution depth. This exploration, however, has the fundamental advantage of precisely and automatically capturing the behaviour of function calls. Moreover, it allows for the implementation of our lazy deployment strategy for smart contracts. Together with our memory-precise verification harnesses, it ensures that function calls are executed with a well-formed `storage` and `memory`. Finally, unlike solc-verify, our approach provides commands to print out the value of variables in a counterexample for debugging purposes.

VeriSol also proposes a Boogie encoding that is checked by Corral. It targets the contracts deployed to the Azure Blockchain and checks for *semantic conformance* of contracts with respect to a workflow policy, which is expressed as a finite-state machine. VeriSol not only finds conformance violations but it derives invariants using predicate abstraction to automatically prove conformance. VeriSol and solc-verify's initial version (in [23]) propose a simple memory model that *does not* account for memory variables. Therefore, it does not capture the different nuances of Solidity's assignment operator, for instance. Their modelling also does not discuss whether function calls and verification harnesses give any well-formedness guarantees for `storage` and `memory` like we do. Moreover, VeriSol does not offer a function harness to quickly find errors that occur deep in the execution of a contract.

6 CONCLUSION

We have combined bounded model checking with a precise memory modelling, lazy contract deployment, and memory-precise verification harnesses to create an innovative verification framework for Solidity smart contracts. Our Boogie encoding accurately captures the semantics of Solidity's memory model, and its lazy contract deployment and memory-precise verification harnesses ensures that contract executions manipulate well-formed state variables and memory cells. Furthermore, our evaluation suggests that Solidifier provides a better speed-precision compromise compared to similar tools.

There are some strong arguments in favour of the approach we propose in comparison to the current state of the art. Firstly, formalising Solidity directly as opposed to EVM bytecode creates a cleaner semantic basis for analysing smart contracts; no need to encode low-level operations and let developers bridge the compilation gap from Solidity to bytecode. Secondly, encoding semantic properties, rather than looking for known vulnerable code patterns, allows the designer of a smart contract to precisely specify what is expected from the contract. Their attempted verification might uncover vulnerabilities that are reachable by unknown behaviour patterns or even function errors; neither of which would be found by traditional tools. Thirdly, bounded model checking systematically, symbolically, and automatically explores all execution paths of a contract up to a certain depth, so it is *guaranteed* to find any error that can occur within the given bound. This exploration tends to exercise all code paths of a smart contract but not all execution paths. On the other hand, current ad-hoc symbolic execution techniques for the analysis of smart contracts typically either too coarsely over-approximate the general behaviour of a smart contract instance, leading to inaccurate violations, or they cover only a small portion of the smart contract's code, missing violations.

Acknowledgements. We thank Gabriela Sampaio, Ante Đerek, Liu Han and the anonymous reviewers for useful feedback on this paper.

REFERENCES

[1] [n.d.]. *Azure blockchain workbench website.* https://github.com/Azure-Samples/blockchain/tree/master/blockchain-workbench/application-and-smart-contract-samples.
[2] [n.d.]. Ethereum White Paper. https://github.com/ethereum/wiki/wiki/White-Paper.

[3] [n.d.]. Hyperledge Fabric website. https://www.hyperledger.org/projects/fabric.

[4] [n.d.]. *Solidifier website*. https://github.com/blockhousetech/research/tree/master/Solidifier.

[5] [n.d.]. *Solidity compiler*. https://github.com/ethereum/solidity.

[6] [n.d.]. *Solidity documentation*. https://solidity.readthedocs.io/.

[7] [n.d.]. *VeriSol repository*. https://github.com/microsoft/verisol.

[8] Wolfgang Ahrendt, Gordon J. Pace, and Gerardo Schneider. 2018. *Smart Contracts: A Killer Application for Deductive Source Code Verification*. Springer International Publishing, Cham, 1–18.

[9] Leonardo Alt and Christian Reitwiessner. 2018. Smt-based verification of solidity smart contracts. In *International Symposium on Leveraging Applications of Formal Methods*. Springer, 376–388.

[10] Sidney Amani, Myriam Bégel, Maksym Bortin, and Mark Staples. 2018. Towards verifying ethereum smart contract bytecode in Isabelle/HOL. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*. ACM, 66–77.

[11] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. 2018. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*. ACM, 30.

[12] Pedro Antonino, Thomas Gibson-Robinson, and A. W. Roscoe. 2019. Efficient Verification of Concurrent Systems Using Synchronisation Analysis and SAT/SMT Solving. *ACM Trans. Softw. Eng. Methodol.* 28, 3 (2019), 18:1–18:43.

[13] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. 2017. A survey of attacks on ethereum smart contracts (sok). In *International Conference on Principles of Security and Trust*. Springer, 164–186.

[14] Christel Baier and Joost-Pieter Katoen. 2008. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press.

[15] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K Rustan M Leino. 2005. Boogie: A modular reusable verifier for object-oriented programs. In *International Symposium on Formal Methods for Components and Objects*. Springer, 364–387.

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

[17] Raymond Cheng, Fan Zhang, Jernej Kos, Warren He, Nicholas Hynes, Noah Johnson, Ari Juels, Andrew Miller, and Dawn Song. 2019. Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 185–200.

[18] Edmund M. Clarke and Jeannette M. Wing. 1996. Formal Methods: State of the Art and Future Directions. *ACM Comput. Surv.* 28, 4 (1996), 626–643.

[19] Michael Crosby, Pradan Pattanayak, Sanjeev Verma, Vignesh Kalyanaraman, et al. 2016. Blockchain technology: Beyond bitcoin. *Applied Innovation* 2, 6-10 (2016), 71.

[20] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. 2018. EtherTrust: Sound static analysis of ethereum bytecode. *Technische Universität Wien, Tech. Rep* (2018).

[21] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. 2018. A semantic framework for the security analysis of ethereum smart contracts. In *International Conference on Principles of Security and Trust*. Springer, 243–269.

[22] Ákos Hajdu and Dejan Jovanović. 2020. SMT-Friendly Formalization of the Solidity Memory Model. In *Programming Languages and Systems*, Peter Müller (Ed.). Springer International Publishing, Cham, 224–250.

[23] Ákos Hajdu and Dejan Jovanović. 2020. solc-verify: A Modular Verifier for Solidity Smart Contracts. In *VSTTE*, Supratik Chakraborty and Jorge A. Navas (Eds.). Springer International Publishing, Cham, 161–179.

[24] Everett Hildenbrandt, Manasvi Saxena, Nishant Rodrigues, Xiaoran Zhu, Philip Daian, Dwight Guth, Brandon Moore, Daejun Park, Yi Zhang, Andrei Stefanescu, et al. 2018. KEVM: A complete formal semantics of the ethereum virtual machine. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*. IEEE, 204–217.

[25] Yoichi Hirai. 2017. Defining the ethereum virtual machine for interactive theorem provers. In *International Conference on Financial Cryptography and Data Security*. Springer, 520–535.

[26] Marco Iansiti and Karim R Lakhani. 2017. The truth about blockchain. *Harvard Business Review* 95, 1 (2017), 118–127.

[27] Harry Kalodner, Steven Goldfeder, Xiaoqi Chen, S Matthew Weinberg, and Edward W Felten. 2018. Arbitrum: Scalable, private smart contracts. In *27th USENIX Security Symposium (USENIX Security 18)*. 1353–1370.

[28] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. ZEUS: Analyzing Safety of Smart Contracts.. In *NDSS*.

[29] Akash Lal, Shaz Qadeer, and Shuvendu K Lahiri. 2012. A solver for reachability modulo theories. In *International Conference on Computer Aided Verification*. Springer, 427–443.

[30] K Rustan M Leino. 2008. This is boogie 2. *manuscript KRML* 178, 131 (2008), 9.

[31] Chao Liu, Han Liu, Zhao Cao, Zhong Chen, Bangdao Chen, and Bill Roscoe. 2018. Reguard: finding reentrancy bugs in smart contracts. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. ACM, 65–68.

[32] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. ACM, 254–269.

[33] Zohar Manna and Amir Pnueli. 1995. *Temporal verification of reactive systems - safety*. Springer.

[34] B Mueller. [n.d.]. Mythril—Reversing and Bug Hunting Framework for the Ethereum Blockchain.

[35] Satoshi Nakamoto et al. 2008. Bitcoin: a peer-to-peer electronic cash system (2008).

[36] Daejun Park, Yi Zhang, Manasvi Saxena, Philip Daian, and Grigore Roşu. 2018. A formal verification tool for ethereum vm bytecode. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 912–915.

[37] Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin Vechev. 2020. Verx: Safety verification of smart contracts. In *2020 IEEE Symposium on Security and Privacy, SP*. 18–20.

[38] Grigore Roşu and Traian Florin Şerbănuţă. 2010. An Overview of the K Semantic Framework. *Journal of Logic and Algebraic Programming* 79, 6 (2010), 397–434.

[39] Ilya Sergey, Amrit Kumar, and Aquinas Hobor. 2018. Temporal Properties of Smart Contracts. In *International Symposium on Leveraging Applications of Formal Methods*. Springer, 323–338.

[40] Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoué, and Santiago Zanella-Béguelin. 2016. Dependent Types and Multi-Monadic Effects in F*. In *43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 256–270.

[41] Melanie Swan. 2015. *Blockchain: Blueprint for a new economy*. "O'Reilly Media, Inc.".

[42] Nick Szabo. 1997. Formalizing and securing relationships on public networks. *First Monday* 2, 9 (1997).

[43] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. 2018. Smartcheck: Static analysis of ethereum smart contracts. In *2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, 9–16.

[44] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. 2018. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 67–82.

[45] Yuepeng Wang, Shuvendu K. Lahiri, Shuo Chen, Rong Pan, Isil Dillig, Cody Born, Immad Naseer, and Kostas Ferles. 2020. Formal Verification of Workflow Policies for Smart Contracts in Azure Blockchain. In *VSTTE*. 87–106.

[46] Karl Wüst, Sinisa Matetic, Silvan Egli, Kari Kostiainen, and Srdjan Capkun. 2019. *ACE: Asynchronous and Concurrent Execution of Complex Smart Contracts*. Technical Report. IACR Cryptology ePrint Archive, 2019: 835, 2019.